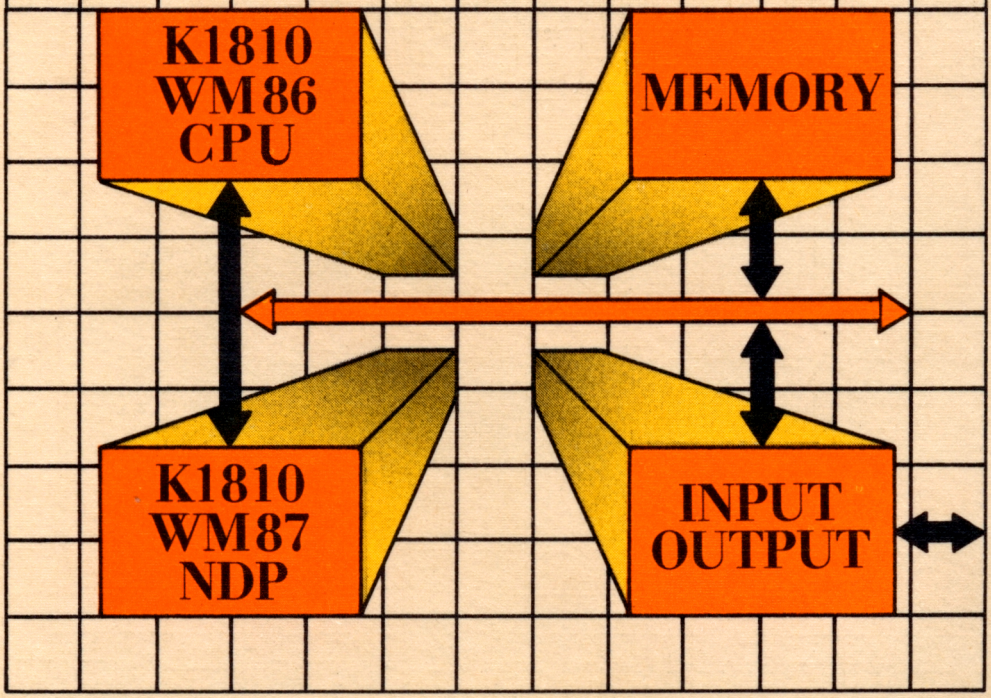
Technische Informatik

Bonitz

Der 16-Bit-Mikroprozessor des ESER-PC



VEB Verlag Technik Berlin

Bonitz

Der 16-Bit-Mikroprozessor des ESER-PC

Technische Informatik

Herausgegeben von

Dr.-Ing. Ludwig Claßen

Dr.-Ing. Dieter Nedo

Dr. rer. nat. Gerhard Paulin

Der 16-Bit-Mikroprozessor des ESER-PC

Dipl.-Phys. Jochen Bonitz

VEB VERLAG TECHNIK BERLIN

Bonitz, Jochen:
Der 16-Bit-Mikroprozessor [Sechzehn-Bit-Mikroprozessor] des ESER-PC / Jochen Bonitz.
– 1. Aufl. – Berlin : Verl. Technik, 1989.
– 204 S. : 33 Abb., 30 Taf. – (Technische Informatik)
ISBN 3-341-00704-0

ISBN 3-341-00704-0
ISSN 0863-0860 Technische Informatik

1. Auflage
© VEB Verlag Technik, Berlin, 1989
Lizenz 201 · 370/83/89
Printed in the German Democratic Republic
Druck und buchbinderische Verarbeitung: IV/10/5 Druckhaus Freiheit Halle
Lektor: HS-Ing. Hans G. Schüler
Einbandgestaltung: Gabriele Schwesinger
LSV 3053 · VT3/6009-1
Bestellnummer: 5540940
02000

Vorwort

Der Personalcomputer ist heute nicht mehr wegzudenken aus Verwaltung, Forschung und Entwicklung. Immer mehr Arbeitsplätze werden umgestaltet; die Mikrorechentechnik hält Einzug in alle Bereiche des gesellschaftlichen Lebens. Ständig werden neue Anwendungsgebiete erschlossen: von Textverarbeitung und Datenbanken über automatisierte Entwurfs- und Fertigungssysteme bis zu modernen Kommunikationstechnologien; auch der Heimbereich darf nicht vergessen werden. Gleichzeitig steigt die Menge der zu verarbeitenden Daten rasch an. Forderungen nach höheren Verarbeitungsgeschwindigkeiten, größerem Speichervolumen und standardisierter, übertragbarer Software sind die Folge.
Der 16-Bit-Mikrocomputer verbindet niedrige Kosten und einfache Handhabung mit dem Leistungsvermögen moderner Kleinrechner. Ein Vertreter der 16-Bit-Generation ist der Mikroprozessor K1810WM86 aus der UdSSR. Er ist funktionell zum internationalen Typ 8086 kompatibel, auf dessen Basis eine große Zahl von Personalcomputern angeboten wird. Diese zeichnen sich durch eine weitestgehende Softwarekompatibilität aus.
Ohne Zweifel steht der 8086 in seiner Architektur den 8-Bit-Mikroprozessoren noch sehr nahe, was ihm im Vergleich mit anderen 16-Bit-Typen, zum Beispiel dem U8000, den Ruf eines leistungsschwachen Vertreters seiner Generation einbrachte. Dennoch war und ist der 8086 kommerziell sehr erfolgreich und wurde - insbesondere in der Klasse der Personalcomputer - zu einem Industriestandard. Worin liegt die Ursache für diesen Erfolg? In erster Linie ist sie in der Verfügbarkeit großer Mengen weitestgehend standardisierter Software zu suchen. Dabei spielte die Möglichkeit eine Rolle, Programme von 8-Bit-Prozessoren zu übertragen und bereits bestehende Ressourcen zu nutzen. Überhaupt ist das Prinzip der Aufwärtskompatibilität als Grundlage einer kontinuierlichen Weiterentwicklung der Mikroprozessortechnik bei dieser Familie besonders ausgeprägt. Es zeichnet sich auch für den nächsten Generationswechsel, die Einführung der 32-Bit-Technik, ab. Die Wiederverwendbarkeit bestehender Software wird auch in Zukunft über den Erfolg neuer Entwicklungen auf dem Gebiet der Mikrorechentechnik entscheiden.

In der DDR wird der K1810WM86 in den Arbeitsplatzcomputern A7100, A7150 und dem Personalcomputer EC1834 eingesetzt. Daneben entstehen zahlreiche Systeme der Industrieautomatisierung, die ebenfalls diese Mikroprozessorfamilie anwenden. Zwangsläufig entsteht ein größer werdendes Bedürfnis nach Informationen über dieses 16-Bit-System. Diesem Anliegen soll der vorliegende Band der Reihe "Technische Informatik" dienen. Es wurde Wert gelegt auf eine möglichst ausführliche Darstellung der Architektur und Arbeitsweise des K1810WM86. Dadurch soll auch dem interessierten Nichtfachmann, der lediglich Grundkenntnisse der Mikrocomputertechnik mitbringt, der Einstieg erleichtert werden. Besonderes Augenmerk wurde auf den Befehlssatz und die Assemblersprache des K1810WM86 gerichtet. Erst durch die genaue Kenntnis der Wirkungsweise eines jeden Befehls ist eine optimale und effektive Assemblerprogram-

mierung möglich. In diesem Sinne wird auch der versierte Soft- und Hardwarespezialist wertvolle Informationen finden.
Ein spezieller Abschnitt ist dem Numerikcoprozessor K1810WM87 gewidmet, der als Gleitkommaprozessor Programme mit hohem Rechenaufwand wirkungsvoll unterstützt. Gerade bei seiner Anwendung bestehen oft Unklarheiten, da kaum Literatur verfügbar ist.
Natürlich kann dieses Buch bei seinem Umfang keine Kenntnisse über den Umgang mit Personalcomputern vermitteln. Die Darstellung wurde bewußt auf den Mikroprozessor beschränkt. Die Befehlsbeschreibungen sind nicht an einen konkreten Assembler gebunden. Damit soll dieser Band nicht nur den Nutzern eines Personalcomputers, sondern allen Anwendern des 16-Bit-Mikroprozessors K1810WM86 eine willkommene Unterstützung bieten.

Der Verfasser möchte dem verantwortlichen Herausgeber, Herrn Dr. L. Claßen, sowie dem VEB Verlag Technik für die Unterstützung und viele wertvolle Hinweise bei der Bearbeitung dieses Bandes danken. Nicht zuletzt gilt der Dank den Mitarbeitern der Entwicklungsabteilung des VEB Numerik "Karl Marx", die durch vielfältige Anregungen und die Hilfe bei der Arbeit an diesem Band zu dessen Gelingen beitrugen, sowie meiner Ehefrau Ramona für ihre verständnisvolle Unterstützung.
Der Autor ist für jeden Hinweis zur Verbesserung der vorliegenden Darstellung dankbar.

Jochen Bonitz

Inhaltsverzeichnis

1. Einführung 11
1.1. Zur Entwicklungsgeschichte des K1810WM86 11
1.2. Grundkonzept des K1810WM86 12

2. Architektur und Arbeitsweise des K1810WM86 15
2.1. Aufbau des K1810WM86 15
2.1.1. Übersicht 15
2.1.2. Signal- und Anschlußbeschreibung 15
2.1.3. Register und Flags 18
2.1.4. Ausführungs- und Businterfaceeinheit 20
2.2. K1810WM86-Bus 23
2.2.1. Buszyklen 23
2.2.2. Busstruktur 25
2.3. Speicherinterface 26
2.3.1. Speicheroperationen 26
2.3.2. Speicherorganisation 28
2.3.3. Byte- und Wortübertragungen 29
2.3.4. Segmentierung und physische Adressen 31
2.3.5. Stapel 33
2.4. Ein-/Ausgabeinterface 34
2.5. Interruptbehandlung 34
2.5.1. Übersicht 34
2.5.2. Interruptvektortabelle 35
2.5.3. Interrupts mit fest zugeordnetem Typ 36
2.5.4. Externe Interruptanforderungen 37
2.5.5. Interne Interrupts 38
2.5.6. Interruptanerkennung 39
2.6. Prozessorsteuerung 41
2.6.1. Reset 41
2.6.2. Busanforderungen 41
2.6.3. Busblockierung 42
2.6.4. Wait und Test 43
2.6.5. Halt-Zustand 43

3. Adressierungsarten 44
3.1. Übersicht 44
3.2. Registeroperanden 46
3.3. Unmittelbare Operanden 46
3.4. Speicheroperanden 46
3.4.1. Datenformate 46
3.4.2. Effektive Adresse 47
3.4.3. Segmentüberschreibung 48
3.4.4. Direkte Adressierung 48
3.4.5. Indirekte Adressierung 49
3.4.6. Relative Adressierung 50
3.4.7. Indizierte Adressierung 51
3.4.8. Basisindizierte Adressierung 51
3.4.9. Stringadressierung 53
3.5. Adressierung von Ein-/Ausgabeports 53
3.6. Generierung der Maschinenbefehle 53

4. Befehlssatz des Mikroprozessors K1810WM86 56
4.1. Übersicht 56
4.2. Datentransferbefehle 57
4.2.1. Transferbefehle für allgemeine Anwendungen ..57
4.2.2. Spezielle Befehle zum Laden von Adressen 60
4.2.3. Befehle zur Flagübertragung 61
4.2.4. Ein-/Ausgabebefehle 62
4.3. Arithmetikbefehle 63
4.3.1. Zahlenformate 63
4.3.2. Flagbeeinflussung 65
4.3.3. Addition 66
4.3.4. Subtraktion 68
4.3.5. Multiplikation 70
4.3.6. Division 72
4.4. Befehle zur Bitmanipulation 74
4.4.1. Flagbeeinflussung 74
4.4.2. Befehle für logische Operationen 75
4.4.3. Bitverschiebe- und Rotationsbefehle 77
4.5. Stringbefehle 80
4.5.1. Adressierung, Register- und Flagbenutzung ...80
4.5.2. Repeat-Präfix 83
4.5.3. Basisoperationen 84
4.6. Programmtransferbefehle 87
4.6.1. Wirkungsweise und Operanden 87
4.6.2. Prozeduraufrufe 88
4.6.3. Sprungbefehle 91
4.6.4. Befehle zur Schleifensteuerung 95
4.6.5. Interruptbefehle 97
4.7. Befehle zur Prozessorsteuerung 98
4.7.1. Befehle zur Flagbeeinflussung 99
4.7.2. Befehle zur externen Synchronisation 100

5. Befehle des K1810WM86 im Überblick 102
5.1. Übersicht 102
5.2. Liste der K1810WM86-Befehle 106

6. Programmierung des K1810WM86 in Assemblersprache 143
6.1. Niveaus der Programmierung 143
6.2. Programmentwicklung 144
6.3. Struktur der Assemblersprache 146
6.3.1. Assembleranweisungen 146
6.3.2. Zahlen und Zeichen 146
6.3.3. Assemblerbefehle 147
6.3.4. Assemblerdirektiven 147
6.4. Vereinbarung von Daten, Segmenten und Prozeduren 148
6.4.1. Symbolzuweisung 148
6.4.2. Datenvereinbarungen 148
6.4.3. Segmentvereinbarungen 150
6.4.4. Prozedurvereinbarung 151
6.4.5. Globale Symbole 152

7. Numerikcoprozessor K1810WM87 153
7.1. Aufbau und Arbeitsweise des K1810WM87 153
7.1.1. Übersicht 153
7.1.2. Hardware-Interface 154
7.1.3. Software-Interface 155
7.1.4. Datentypen und Formate 156
7.1.5. Register 158
7.1.6. Busoperationen 162
7.2. Programmierung des K1810WM87 163
7.2.1. Adressierungsarten 163
7.2.2. Spezielle Werte 163
7.2.3. Befehlssynchronisation mit WAIT 165
7.2.4. Exception-Behandlung 166
7.2.5. Interrupts 168
7.2.6. Initialisierung 169
7.3. Befehlssatz des K1810WM87 170
7.3.1. Datentransferbefehle 171
7.3.2. Arithmetikbefehle 172
7.3.3. Vergleichsbefehle 175
7.3.4. Transzendente Funktionen 176
7.3.5. Befehle zum Laden von Konstanten 178
7.3.6. Befehle zur Prozessorsteuerung 178

Anhang A: Befehle des K1810WM86 182
Anhang B: Befehle des K1810WM87 188
Anhang C: Maschinenbefehle in numerischer Reihenfolge 192

Literaturverzeichnis 201
Sachwörterverzeichnis 202

1. Einführung

Mit dem 16-Bit-Mikroprozessor K1810WM86 wurde eine neue Etappe der Computerentwicklung eingeleitet. Dieses Buch beschreibt den K1810WM86 - seinen Aufbau, die Arbeitsweise und Programmierung. Ein spezieller Abschnitt widmet sich dem Numerikcoprozessor K1810WM87, der Anwendungen mit hohem Rechenaufwand wirkungsvoll unterstützt. Zunächst sollen die Grundlagen der Entwicklung der K1810WM86-Familie betrachtet werden.

1.1. Zur Entwicklungsgeschichte des K1810WM86

Mit den 16-Bit-Mikroprozessoren hat eine neue Computergeneration weltweit Eingang in Industrie und Forschung gefunden. Der K1810WM86 kann 1 MByte Speicher adressieren - dagegen nehmen sich die 64 KByte der verbreitetsten 8-Bit-Mikroprozessoren recht bescheiden aus. Deshalb ist auch für Personalcomputer der 16-Bit-Mikroprozessor zur Selbstverständlichkeit geworden, müssen doch ständig größere Datenmengen beherrscht werden.
Blickt man zurück, so erkennt man - nach der Geburtsstunde des "Elektronenhirns" und einer Periode, die vom Bau mächtiger Rechenmaschinen geprägt war - eine Spaltung des Entwicklungsweges von Computern etwa um das Jahr 1970. Die klassischen Computer entwickelten sich zu Großrechnern mit 32 oder 64 Bit Datenbreite, die immer kompliziertere Rechnungen immer schneller ausführen sollten. Der Preis waren Schränke voller teuerster Elektronik, die für eine Massenanwendung ungeeignet waren.
Der andere Entwicklungsweg wurde durch die Mikroelektronik eingeleitet und geprägt. Kleine Ausmaße und niedrige Preise - das waren die Vorraussetzungen für eine breitere Anwendung. Innerhalb von nur drei Jahren wurde vom ersten Mikroprozessor 4004 (4 Bit) über den 8008 (8 Bit) zum 8080 das Fundament für die Massenanwendung des Mikrocomputers gelegt. Die nächsten 10 Jahre gehörten dem 8-Bit-Mikroprozessor. In der Architektur des U880 erreichte diese Generation wohl ihren Höhepunkt. Auch heute hat der 8-Bit-Prozessor noch seine Berechtigung, vor allem als Einchip-Computer für Konsumgüter und einfache industrielle Anwendungen.
Bei der Verbesserung der 8-Bit-Systeme zeigte sich sehr bald, daß Mikrocomputer ihren größeren und älteren Verwandten ebenbürtig sein können. Ein 16-Bit-Computer auf einem Chip kann völlig neue Dimensionen erschließen, in die Welt der Großrechner eindringen.
Vor 10 Jahren gelang dieser Schritt mit dem 8086 der Firma Intel, und seitdem ist dieser Mikroprozessor zu einem Industriestandard geworden. (Kurz vorher hatte Texas Instruments den ersten 16-Bit-Prozessor vorgestellt.) Andere Typen folgten, mit teilweise besseren Leistungsmerkmalen, so auch der U8000, der erste 16-Bit-Mikroprozessor der DDR /1/. Doch die weltweite Verbreitung, das umfangreiche Softwareangebot und das Konzept der Aufwärtskompatibilität mit neuen Entwicklungen verhalfen dem 8086 zu seiner großen Popularität /2/. Mit diesem Typ wurde ein nahtloser Übergang von der 8-Bit- zur 16-Bit-Technik erreicht.

In der UdSSR wurde mit dem Typ K1810WM86 ein 16-Bit-Mikroprozessor entwickelt, der zum 8086 kompatibel ist /3/. Natürlich sind international längst Weiterentwicklungen dieser Familie auf dem Markt, und auch in der DDR werden neue 16-Bit-Prozessoren entwickelt. Vorerst wird jedoch der K1810WM86 dominieren.
Die Personalcomputer A7100, A7150, der ESER-PC EC1834 wie auch zahlreiche Systeme der Industrieautomatisierung verwenden diesen Mikroprozessor /4/ /5/.

1.2. Grundkonzept des K1810WM86

Was verbirgt sich hinter dem Begriff "16-Bit-Mikroprozessor"? Die Prozessoren der 8-Bit-Generation, etwa der U880, verarbeiten Daten mit einer Breite von 8 Bit. Solche als Byte bezeichneten Dateneinheiten können 256 verschiedene Zustände repräsentieren. Der K1810WM86 verarbeitet Daten mit einer Breite von 16 Bit. Ein solches, aus zwei Byte bestehendes Wort kann 65536 verschiedene Zustände darstellen. Der K1810WM86 kann solche Daten nicht nur intern verarbeiten, sondern auch zu oder von angeschlossenen Speichern oder Ein-/Ausgabegeräten übertragen. Tafel 1.1 zeigt einige weitere Unterschiede zum U880.
Um zu bestehenden 8-Bit-Systemen kompatibel zu sein, wurde parallel zum K1810WM86 noch der K1810WM88 entwickelt. Dabei handelt es sich um einen Prozessor, der intern wie der K1810WM86 arbeitet, Daten aber nur auf einem 8-Bit-Bus übertragen kann. Es handelt sich also hierbei um einen "abgerüsteten" 16-Bit-Mikroprozessor, der über den gleichen Befehlssatz wie sein "größerer Bruder" verfügt. Der Programmierer merkt keinen Unterschied. Da einerseits Programme von 8-Bit-Prozessoren übertragbar waren und andererseits bestehende Hardware-Peripherie nutzbar blieb, wurde ein problemloser Übergang möglich. Diese Strategie trug wesentlich zum Erfolg des K1810WM86 bei. Ein ähnlicher Weg zeichnet sich heute schon bei der Einführung der 32-Bit-Technik ab.

Tafel 1.1. Einige Unterschiede zwischen K1810WM86 und U880

Parameter	K1810WM86	U880
Datenbreite	16 Bit	8 Bit
Speicheradreßraum	1 MByte	64 KByte
Ein-/Ausgabeadreßraum	64 KByte	256 Byte
Taktfrequenz	5 oder 8 MHz	2,5 oder 4 MHz
Registersatz	14 x 16 Bit	16 x 8 Bit + 4 x 16 Bit
Befehlssatz	ca.300 Befehle	158 Befehle

Aus der Entwicklungsgeschichte der 16-Bit-Mikroprozessoren kennen wir das Ziel, das beim Entwurf des K1810WM86 verfolgt wurde - Einstieg in die Welt der Minicomputer. Diesem Ziel ordnet sich die fundamentale Architektur des K1810WM86 unter.

Obwohl noch viele Ähnlichkeiten mit den 8-Bit-Mikroprozessoren bestehen, wurden einige völlig neue Konzepte verwirklicht, die den K1810WM86 als erstes Produkt einer neuen Generation von Mikroprozessoren charakterisieren. Diese Grundkonzepte sind:

Speichersegmentierung
Das Konzept der Speichersegmentierung bildet die Voraussetzung für eine modulare Programmierung. Jeder Programmodul kann gleichzeitig mit vier Segmenten operieren:

- ein Kodesegment für das Programm,
- ein Datensegment für Speicherdaten,
- ein Stapelsegment für die Koordinierung der Zusammenarbeit mehrerer Module zur Lösung einer Aufgabe (Task),
- ein Extradatensegment für den Datenaustausch zwischen verschiedenen Modulen.

Jedes Segment ist 64 KByte groß und kann überall im 1-MByte-Speicherraum lokalisiert werden. Unter Softwarekontrolle können diese Segmente dynamisch verschoben werden.

Komplexe Adressierung von Daten
Die Adressierung von Speicherdaten erfolgt durch vier Komponenten:

- Segmentbasisadresse, welche die Lage des 64-KByte-Segments im 1-MByte-Adreßraum bestimmt,
- Basiszeiger zur dynamischen Auswahl von Datenstrukturen im Segment,
- Index zur dynamischen Auswahl von Elementen in der Struktur,
- Displacement als konstanter Adreßanteil (Abstand).

Innerhalb der Segmente wird nur mit 16-Bit-Adressen operiert. Für alle Operanden berechnet der K1810WM86 eine effektive Adresse aus der Summe von Basis, Index und Displacement. Die Segmentbasis wird automatisch hinzugefügt, um eine 20-Bit-Adresse zu erhalten. Dieses Adressierungsverfahren kommt den von höheren Programmiersprachen unterstützten Datenstrukturen entgegen.

Registerstruktur
Der K1810WM86 kann bei seinen Operationen auf einen umfangreichen Registersatz zurückgreifen:

- vier universelle 16-Bit-Datenregister, die bei Bedarf in acht 8-Bit-Register geteilt werden können,
- vier Zeiger- und Indexregister, die auch als 16-Bit-Datenregister verwendet werden können,
- vier Segmentregister; sie enthalten die 16-Bit-Basisadressen der Segmente, auf die implizit zugegriffen wird.

Dazu kommen noch ein Befehlszeiger für die Programmadresse und ein Programmstatuswort, welches Ergebnis- und Steuerflags enthält.

Optimierter Befehlssatz
Der Befehlssatz des K1810WM86 ist für effektive Speicherauslastung und hohe Ausführungsgeschwindigkeit ausgelegt. Die Mehrzahl der Befehle kann sowohl Byte- als auch Wortdaten verarbeiten. Die Befehle unterstützen sechs Gruppen von Operationen:

- Datentransfer,
- Arithmetik,
- Logik,
- Stringmanipulation,
- Programmtransfer,
- Steuerung.

Hervorzuheben ist die Möglichkeit der Befehlssatzerweiterung durch den ESC-Befehl ("Escape"). Dieser erlaubt die Generierung von 64 zusätzlichen Kodes, die von externen Prozessoren (Coprozessoren) als Befehle interpretiert und ausgeführt werden können. Zur Programmierung stehen leistungsfähige Assembler zur Verfügung. Die Assemblersprache enthält etwa 100 verschiedene Mnemoniks für die K1810WM86-Befehle. Direktiven für Datendefinitionen, Speicher- und Prozedurvereinbarungen unterstützen eine modulare, übersichtliche und hardwarenahe Programmierung /6/ /7/.
Natürlich können auch Programme in höheren Programmiersprachen geschrieben werden. Für verschiedene Betriebssysteme sind BASIC-Interpreter und Compiler für die Sprachen PL/M, FORTRAN, PASCAL, C und andere verfügbar.

Unterstützung für Multiprozessorsysteme
Die Familie des K1810WM86 bietet eine umfangreiche Unterstützung für den Aufbau leistungsfähiger Multiprozessorsysteme, in denen mehrere Prozessoren parallel arbeiten. Dazu gehören auch die Coprozessoren der K1810-Familie:

- Numerikcoprozessor K1810WM87
  Dieser Coprozessor erlaubt die effektive Ausführung arithmetischer Operationen. Mit seiner Hilfe kann die Rechengeschwindigkeit um einen Faktor bis zu 100 gesteigert werden. Zur Verarbeitung von Daten im Dezimal-, Integer- und Real-Format stehen acht 80-Bit-Register und 69 zusätzliche Befehle zur Verfügung.

- Ein-/Ausgabeprozessor K1810WM89
  Er entlastet den K1810WM86 von zeitintensiven Datenübertragungen. Datentransfers von oder zu Ein-/Ausgabegeräten oder zwischen Speichermodulen führt er mit einer Datenrate bis zu 1,25 MByte je Sekunde aus. Er verfügt über einen eigenen Befehlssatz und kann autonom unter der Kontrolle des K1810WM86 arbeiten.

Diese Grundprinzipien erlauben den Einsatz des K1810WM86 für ein großes Spektrum von Anwendungen - vom Einplatinenrechner bis zu komplizierten Mehrrechnerstrukturen. In den folgenden Abschnitten wollen wir uns mit der Architektur, der Arbeitsweise und der Programmierung vertraut machen.

2. Architektur und Arbeitsweise des K1810WM86

Dieser Abschnitt gibt einen Überblick über Architektur und Arbeitsweise dieses Mikroprozessors der dritten Generation. Es werden die inneren Abläufe und das Zusammenwirken mit Speicher und Ein-/Ausgabegeräten erläutert. Auf detaillierte elektrische Parameter und Zeitabläufe soll hier verzichtet werden.

2.1. Aufbau des K1810WM86

2.1.1. Übersicht

Der K1810WM86 ist ein echter 16-Bit-Prozessor. Er kann nicht nur Daten im 16-Bit-Format verarbeiten, sondern auch über den Bus übertragen. Zur Unterstützung unterschiedlicher Anwendungen kann der K1810WM86 in zwei Betriebsarten arbeiten:

Minimum-Mode: Für kleine Systeme, z.B. Einplatinenrechner, erzeugt der Prozessor alle Steuersignale für das Zusammenwirken mit Speicher- und Ein-/Ausgabebausteinen selbst.

Maximum-Mode: Für mittlere und größere Systeme, die mehrere Prozessoren enthalten können und deren Module über ein Bussystem verbunden sind, erzeugt der K1810WM86 verschiedene Statussignale, die in externen Steuerbausteinen verarbeitet werden. Diese bilden ihrerseits die geforderten Steuersignale.

Beim Hardwareentwurf muß eine der Betriebsarten ausgewählt werden. (Meist wird der Maximum-Mode angewendet.)

2.1.2. Signal- und Anschlußbeschreibung

Im Bild 2.1 ist die Anschlußbelegung des K1810WM86 dargestellt. Anschluß 33 legt die Betriebsart fest:

- Bei H-Pegel am Anschluß 33 arbeitet der Prozessor im Minimum-Mode; dabei erhalten die Anschlüsse 24 bis 31 die in Klammern angegebene Bedeutung.

- Bei L-Pegel am Anschluß 33 arbeitet der Prozessor im Maximum-Mode.

Die Tafel 2.1 enthält eine Beschreibung aller Signale.
Dabei wird folgende Notation verwendet:

- Signale mit "/" vor dem Namen sind bei L-Pegel aktiv,
- Signale ohne Kennzeichnung sind bei H-Pegel aktiv.

(L = TTL-low, H = TTL-high)

Tafel 2.1. Signale des K1810WM86

Name	Funktion	Typ
	A l l g e m e i n e S i g n a l e	
VCC	Speisespannung, +5V	
GND	Bezugspotential, Masse	
CLK	Takt (5 oder 8 MHz)	E
MIN/MAX	Modeauswahl (Minimum / Maximum)	E
AD0-AD15	gemultiplexter 16-Bit-Adreß-/Datenbus	E/A,TRI
A16-A19	Adreßbits A16-A19,	A,TRI
/BHE	Freigabe des höherwertigen Bytes	A,TRI
S3-S7	Statussignale 3 bis 7 (mit A16-19 und /BHE gemultiplext)	A,TRI
RESET	Rücksetzen (Initialisieren)	E
READY	Quittung für Buszyklus	E
/TEST	Testeingang für WAIT-Operation	E
INTR	Maskierbare Interruptanforderung	E
NMI	Nichtmaskierbare Interruptanforderung	E
/RD	Lesekommando	A,TRI
S i g n a l e i m M i n i m u m - M o d e (MIN/MAX = VCC)		
/WR	Schreibkommando	A,TRI
M/IO	Speicher- oder Ein-/Ausgabeoperation	A,TRI
/INTA	Interruptanerkennung	A,TRI
/DEN	Datentreiberfreigabe	A,TRI
DT/R	Datentreibersteuerung (Senden/ Empfangen)	A,TRI
ALE	Adreßlatchfreigabe (Adresse gültig)	A
HOLD	Busanforderung	E
HOLDA	Anerkennug der Busanforderung	A
S i g n a l e i m M a x i m u m - M o d e (MIN/MAX = GND)		
/S0-/S2	Status des Buszyklus	A,TRI
/LOCK	Blockierung der Buszugriffssteuerung	A,TRI
/RQ/GT0, /RQ/GT1	Anforderung / Gewährung der Bus-steuerung	E/A
QS0,QS1	Status der Befehlswarteschlange	A
Erläuterung der Typen:	E - Eingang, A - Ausgang, E/A - bidirektionales Signal, TRI - Tri-State-Signal.	

Zur Reduzierung der Anschlußzahl verfügt der K1810WM86 über einen gemultiplexten Adreß-/Datenbus. Die Signale AD0-AD15 führen am Beginn eines Buszyklus die Adreßinformation (Adreßbits 0 bis 15), anschließend werden auf ihnen die Daten übertragen.
Die Funktion der Signale wird in den folgenden Abschnitten beschrieben.

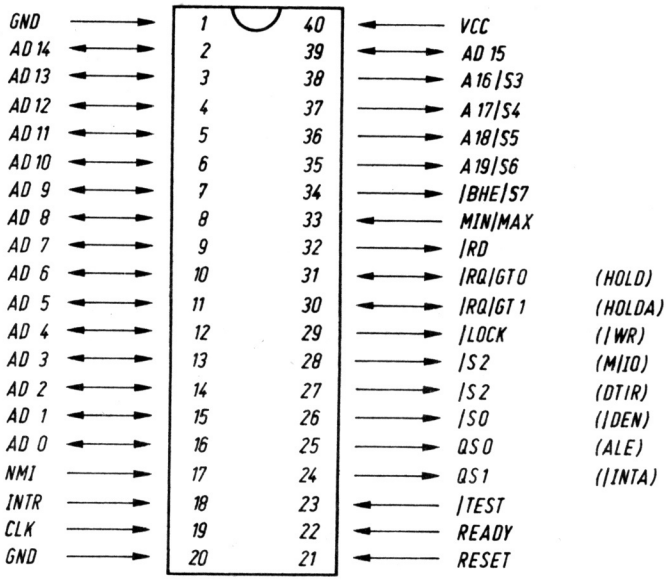


Bild 2.1. Anschlußbelegung des K1810WM86

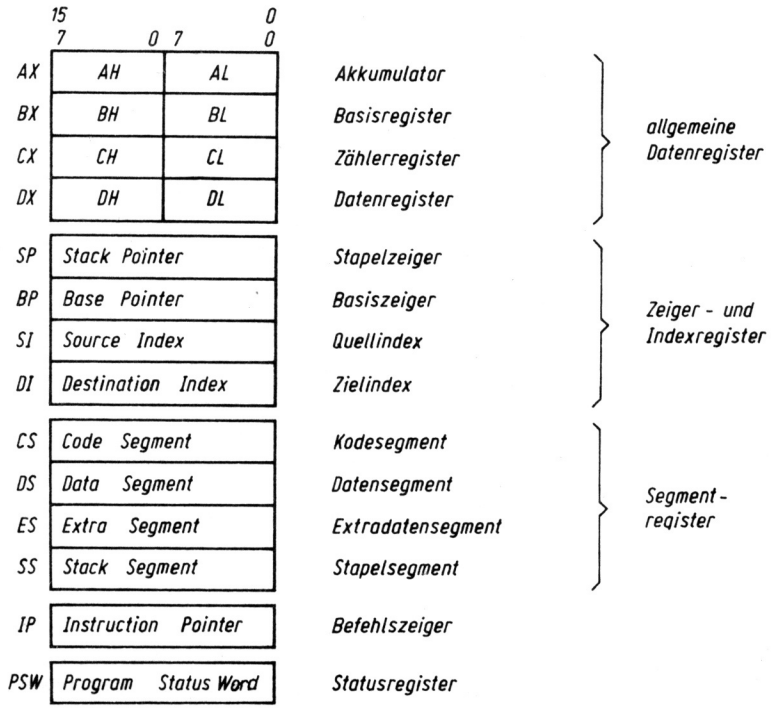


Bild 2.2. Registersatz des K1810WM86

2.1.3. Register und Flags

Der K1810WM86 enthält 14 Register (Bild 2.2). Davon können acht als Datenregister verwendet werden. Das sind interne, 16 Bit breite Speicherplätze, auf die besonders schnell zugegriffen werden kann. Die Register des K1810WM86 werden in fünf Gruppen eingeteilt:

Datenregister
Die vier 16-Bit-Datenregister heißen AX, BX, CX und DX. Sie werden bei Operationen mit 16-Bit-Operanden verwendet. Jedes dieser Register kann bei Bedarf in zwei 8-Bit-Register geteilt werden. Diese heißen dann AL, BL, CL, DL und AH, BH, CH, DH, je nachdem, ob sie das niederwertige (low) oder höherwertige (high) Byte repräsentieren. Diese acht Register können für Byteoperationen genutzt werden, die Ausführungszeit und Speicherplatz sparen. Alle Register können als allgemeine Datenregister verwendet werden, jedoch verlangen einige Befehle implizit die Anwendung bestimmter Register:

AX wird als Akkumulator benutzt bei Multiplikationen und Divisionen, bei Ein-/Ausgaben und bei einigen Stringoperationen.
AL wird als Akkumulator bei den entsprechenden Byteoperationen verwendet.
CX wird als Schleifenzähler bei Schleifenbefehlen und bei Stringoperationen mit Wiederholungspräfix (Repeat) benutzt.
CL wird als Zähler für Bitverschiebebefehle verwendet.
DX enthält die Portadresse bei Ein-/Ausgabeoperationen und wird bei Wortmultiplikationen und -divisionen als Akkumulatorerweiterung benötigt.

Viele Befehle generieren besonders kurze Maschinenkodes, wenn der Akkumulator als Operand verwendet wird.

Zeiger- und Indexregister
Die beiden Zeigerregister SP und BP enthalten die Offsetadressen für Operationen im Stapelsegment. Die Indexregister SI und DI enthalten bei Stringoperationen die Offsetadressen für das Datensegment und das Extradatensegment.
Neben diesen speziellen Anwendungen können diese vier 16-Bit-Register als allgemeine Datenregister genutzt werden.

Segmentregister
Die vier Segmentregister CS, DS, ES, SS enthalten die aktuellen Basisadressen für die vier Speichersegmente, mit denen der K1810WM86 gleichzeitig arbeiten kann. Sie dienen der Speicheradressierung.
Innerhalb eines jeden Segmentes werden die Speicherzellen durch ihren Abstand von der Basisadresse adressiert. Dieser Abstand wird als Offsetadresse bezeichnet. Wie mit Hilfe der im Segmentregister enthaltenen Basisadressen und der in anderen Registern enthaltenen Offsetadressen die wirklichen Speicheradressen gebildet werden, ist im Abschnitt 2.3.4. ausführlich beschrieben.

Die jeweils 64 KByte großen Segmente sind:

CS **Kodesegment**
Dieser Speicherbereich enthält das aktuelle Programm, welches gerade abgearbeitet wird.

DS **Datensegment**
Es enthält die Daten des aktuellen Programms.

ES **Extradatensegment**
Dieses Segment enthält gleichfalls Daten und kann wie DS benutzt werden. Bei Stringoperationen werden Daten von DS nach ES transportiert.

SS **Stapelsegment**
Dieses spezielle Datensegment bildet den Stapel (Stack) für das aktuelle Programm. Es dient dem Zwischenspeichern von Daten bei Unterprogrammaufrufen und Interrupts.

Befehlszeiger
Dieses Register (IP) hat eine ähnliche Funktion wie der Programmzähler bekannter 8-Bit-Mikroprozessoren. Es enthält die Offsetadresse des Befehls, der als nächster aus dem aktuellen Kodesegment gelesen wird. Beim K1810WM86 ist jedoch dieser Befehl meistens nicht identisch mit dem Befehl, der als nächster abgearbeitet wird, da dieser über die Fähigkeit verfügt, Befehle im voraus einzulesen ("Prefetching"), das heißt, das Befehlslesen ist der Ausführung stets um einige Byte voraus.
Immer, wenn IP in den Stapel übertragen wird, wird es aktualisiert, das heißt, im Stapel ist immer der Offset des nächsten auszuführenden Befehls abgelegt.

Programmstatuswort
Dieses Register (PSW) enthält die Kennzeichen oder Flags. Das sind einzelne Bits, die im Ergebnis der ausgeführten Operation gesetzt, zurückgesetzt oder nicht beeinflußt werden. Je nach Zustand dieser Flags trifft der Prozessor bestimmte Entscheidungen, die vom Programm gefordert werden.
Das PSW hat folgendes Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Flag	R	R	R	R	O	D	I	T	S	Z	R	A	R	P	R	C

Die mit R gekennzeichneten Bits sind dem Programmierer nicht zugänglich und beim Lesen des PSW auf 0 gesetzt.
Sechs Bits sind Ergebnisflags:

O **Überlaufflag** (Overflow Flag)
Dieses Bit wird auf 1 gesetzt, wenn bei arithmetischen Operationen mit vorzeichenbehafteten Zahlen betragsmäßig zu große Operanden verwendet wurden, so daß das Ergebnis das Format des Zieloperanden überschreitet. Außerdem zeigt es in Verbindung mit C die Länge des Multiplikationsergebnisses an: Sind beide Bits gesetzt, so ist das höherwertige Byte oder Wort ungleich 0; haben beide Bits den Wert 0, so ist auch der höherwertige Teil des Ergebnisses gleich 0.

S **Vorzeichenflag** (Sign Flag)
Dieses Bit wird auf 1 gesetzt, wenn eine Operation mit vorzeichenbehafteten Zahlen ein negatives Ergebnis produziert.

Z **Nullflag** (Zero Flag)
Wenn das Ergebnis einer Operation 0 ist, wird dieses Bit auf 1 gesetzt.

A **Hilfsübertragsflag** (Auxiliary Carry Flag)
Dieses Bit dient der Verarbeitung von BCD-Zahlen. Es wird auf 1 gesetzt, wenn ein Übertrag aus dem niederwertigen Halbbyte entsteht.

P **Paritätsflag** (Parity Flag)
Enthält das Ergebnis einer Operation eine gerade Zahl von Bits, die den Wert 1 haben, so wird dieses Bit auf 1 gesetzt (gerade Parität).

C **Übertragsflag** (Carry Flag)
Dieses Bit wird auf 1 gesetzt, wenn im Ergebnis einer Operation ein Übertrag an der höchsten Bitposition entsteht.

Drei weitere Flags dienen der Programmsteuerung und können unmittelbar mit Befehlen beeinflußt werden:

D **Richtungsflag** (Direction Flag)
Dieses Bit wird mit dem Befehl STD auf 1 gesetzt und mit CLD rückgesetzt. Es wird bei Stringoperationen benutzt, um anzugeben, ob die Indexregister abwärts (D=1) oder aufwärts (D=0) gezählt werden sollen.

I **Interruptflag** (Interrupt Flag)
Mit dem Befehl STI kann dieses Bit auf 1 gesetzt, mit CLI rückgesetzt werden. Der Prozessor akzeptiert maskierbare Interruptanforderungen nur, wenn dieses Bit gesetzt ist.

T **Testflag** (Trap Flag)
Dieses Bit kann nur durch Manipulation im Stapel gesetzt werden. (Ein Beispiel hierzu ist im Abschn. 4 bei der Beschreibung des Befehls POPF zu finden.) Anschließend geht der Prozessor für Testzwecke in den Einzelschrittmodus, der im Abschnitt 2.5.3 beschrieben wird.

2.1.4. Ausführungs- und Businterfaceeinheit

Ein Mikroprozessor arbeitet Befehle nach folgendem Schema ab:

1. Befehlskode lesen
2. Operand lesen (falls erforderlich)
3. Befehl ausführen
4. Ergebnis schreiben (falls erforderlich)

Die Prozessoren der 2.Generation arbeiten dieses Schema sequentiell ab, das heißt, sie führen jeweils den Befehl aus, den sie gerade gelesen haben, und erst nach beendeter Ausführung wird der nächste gelesen. Nach außen ist dieser Prozessor dabei nur während der Lese- oder Schreiboperationen aktiv, während der Ausführungsphase arbeitet er nur intern. Diese passiven Zustände können dabei beträchtliche Zeitanteile einnehmen, beispielsweise bei arithmetischen Operationen.
Bei der Entwicklung des K1810WM86 hat man nach Wegen gesucht, diese "Leerzeiten" besser auszunutzen, indem während der Befehlsausführung schon ein oder zwei weitere Befehle auf Vorrat eingelesen werden. Erreicht wurde dies durch eine sehr effektive Struktur. Der Prozessor ist in zwei voneinander unabhängig arbeitende Einheiten geteilt:

- Ausführungseinheit (EU),
- Businterface-Einheit (BIU).

Im Bild 2.3 ist diese Struktur dargestellt, während Bild 2.4 schematisch die Arbeitsweise dieser Struktur demonstrieren soll.

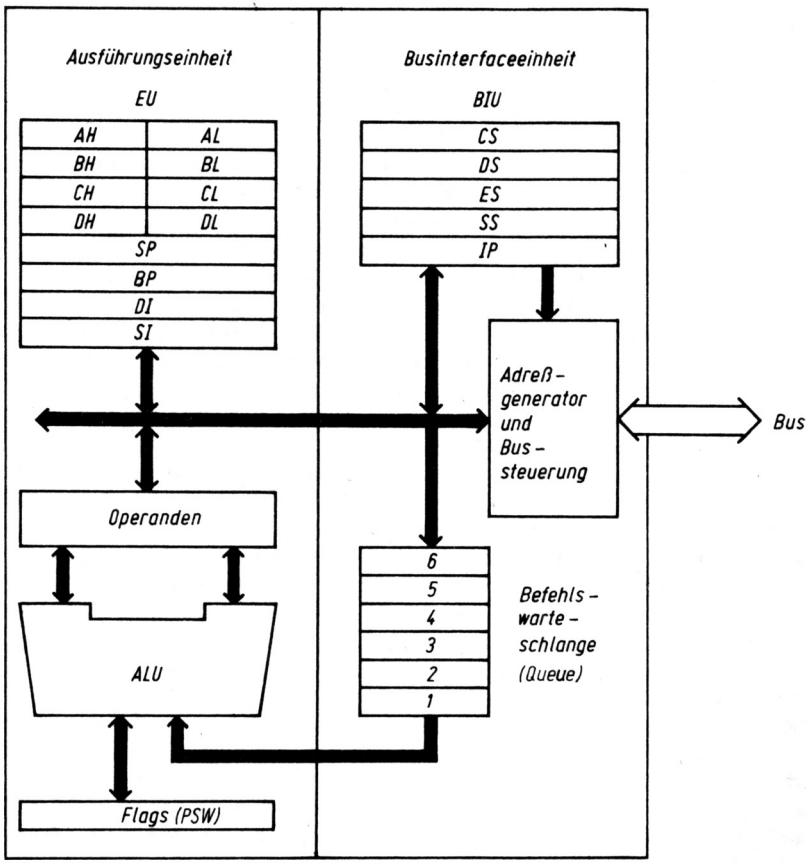
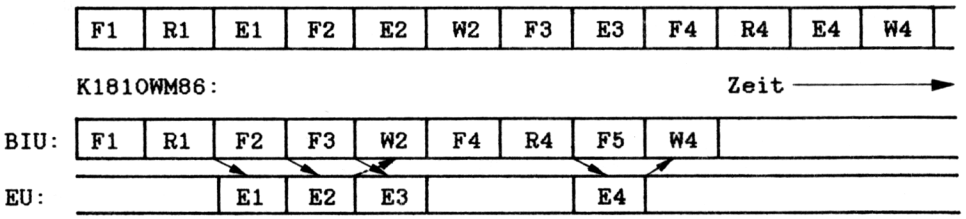


Bild 2.3. Struktur des K1810WM86

Prozessor der 2.Generation:



F = Kodelesen (Fetch) R = Daten lesen (Read)
E = Ausführen (Execute) W = Daten schreiben (Write)

Bild 2.4. Arbeitsweise von BIU und EU

Anhand dieses Beispiels wird die Steigerung der Arbeitsgeschwindigkeit deutlich. In Wirklichkeit sind die Vorgänge natürlich komplizierter, deshalb soll die Wirkungsweise der beiden Komponenten noch etwas genauer betrachtet werden.

Ausführungseinheit (Execution Unit):
Sie enthält alle 8 Datenregister und die Ausführungslogik. Die EU entnimmt einen Befehl aus dem Koderegister und führt ihn aus. Sind zu diesem Zweck Busoperationen erforderlich, etwa das Lesen von Operanden oder eine Ein-/ Ausgabeoperation mit einem Gerätekanal, so informiert sie die BIU. Diese berechnet die Adresse und generiert den erforderlichen Buszyklus, während die EU wartet. Sie kennt keine Zyklen, sondern führt nur aus oder wartet, daß die BIU Operanden überträgt. Die Ausführung der Befehle dauert jeweils eine unterschiedliche Anzahl von Takten. Ist ein Befehl abgearbeitet, wird die BIU veranlaßt, das Ergebnis auszugeben, während die EU schon den nächsten Befehl aus dem Koderegister holt. Die BIU hat nämlich in einer Zeit, in der sie nicht von der EU gebraucht wurde, neue Befehlsbytes geholt.

Businterfaceeinheit (Bus Interface Unit):
Die BIU enthält die 4 Segmentregister und den Adreßgenerator zur Bildung der physischen Adressen, außerdem den Befehlszeiger und ein 6-Byte-Register, welches Befehlswarteschlange (Instruction Queue) genannt wird. Die BIU ist verantwortlich für die Kommunikation mit der "Außenwelt" des Prozessors. Auf Anforderung durch die EU generiert sie Speicher- und Ein-/Ausgabezyklen. Ist die EU jedoch beschäftigt und hat keine Aufträge für die BIU, sorgt diese dafür, daß die Befehlswarteschlange stets gefüllt ist. Immer, wenn mindestens zwei Byte frei sind, generiert die BIU selbständig einen Befehlskode-Lesezyklus. Die Warteschlange ist also stets mit Befehlen von fortlaufenden Speicherplätzen aus dem Kodesegment gefüllt, und die EU muß in der Regel nicht warten. Das bedeutet, daß die Befehlslesezeiten bei der Berechnung von Programmlaufzeiten bei geradlinigen, also unverzweigten Programmen nicht berücksichtigt werden müssen; sie werden praktisch eingespart. Eine Ausnahme bilden die Programmverzweigungen, die

beispielsweise durch Sprungbefehle oder Interrupts entstehen. In diesem Fall fordert die EU einen neuen Befehlskode an. Daraufhin löscht die BIU die Warteschlange und füllt sie wieder mit Bytes aus dem neuen Adreßbereich.

2.2. K1810WM86-Bus

2.2.1. Buszyklen

Der zeitliche Ablauf einer Operation, während der eine Speicherzelle oder ein Ein-/Ausgabekanal adressiert und ein Byte oder Wort übertragen wird, heißt Buszyklus.

Nach der Art des Adreßraumes, der bei der Operation angesprochen wird, unterscheidet man:

- Speicheroperationen (M/IO bzw. /S2 = H),
- Ein-/Ausgabeoperationen (M/IO bzw. /S2 = L).

Nach der Richtung des Datenflusses unterscheidet man dabei:

- Leseoperationen (Kode oder Daten aus dem Speicher lesen, Eingabeport lesen),
- Schreiboperationen (Daten in den Speicher schreiben, Ausgabeport beschreiben)

Eine weitere Operation, die wir später noch kennenlernen, ist die Interruptanerkennung. In ihrem Zeitverhalten entspricht sie einem Lesezyklus, allerdings wird statt des /RD-Signals das Signal /INTA aktiv. Die Tafel 2.2 zeigt die möglichen Buszustände.
Den prinzipiellen Signalverlauf für einen Buszyklus zeigt Bild 2.5. (Dargestellt ist der Minimum-Mode; der Maximum-Mode unterscheidet sich bei Einsatz entsprechender Steuerschaltkreise, z.B. K1810WG88, nicht wesentlich.)

Tafel 2.2. Buszustände des K1810WM86

Buszustand	Maximum-Mode /S2	/S1	/S0	Minimum-Mode M/IO	/RD	/WT	/INTA
INT-Anerkennung	L	L	L	L	H	H	L
E/A-Port lesen	L	L	H	L	L	H	H
E/A-Port schreiben	L	H	L	L	H	L	H
HALT	L	H	H	L	H	H	H
Kode lesen	H	L	L	H	L	H	H
Daten lesen	H	L	H	H	L	H	H
Daten schreiben	H	H	L	H	H	L	H
Passiv	H	H	H	H	H	H	H

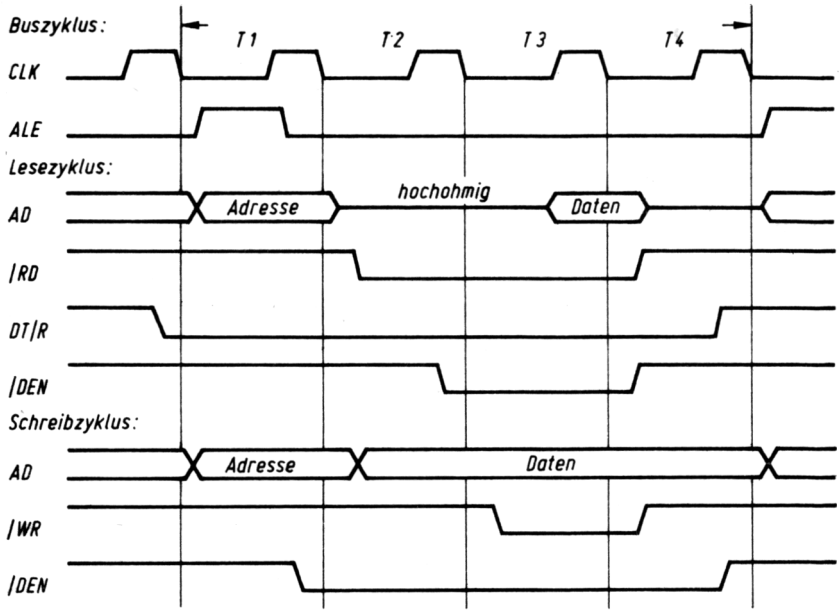


Bild 2.5. Buszyklus des K1810WM86

Jeder Buszyklus besteht aus vier Takten:

T1: Ausgabe der Adresse,
T2: Ausgabe des Kommandosignals,
bei Schreiboperationen Ausgabe der Daten,
T3: bei Leseoperationen Übernahme der Daten,
T4: Kommandosignal abschalten,
bei Schreiboperationen Abschalten der Schreibdaten.

Nicht alle adressierten Bausteine sind in der Lage, innerhalb der durch dieses Regime vorgegebenen Zeiten Daten auszusenden oder zu übernehmen. Deshalb verfügt der Prozessor noch über einen Mechanismus, den Buszyklus zu verlängern. Zu Beginn des T3-Taktes wird der Eingang READY getestet, der von den adressierten Einheiten beeinflußt werden kann. Ist dieses Signal aktiv, so wird der Datentransfer durchgeführt und der Zyklus beendet. Falls READY jedoch nicht aktiv ist, wird der Buszustand beibehalten, und der Prozessor fügt Wartetakte TW ein. Diese entsprechen den T3-Takten, das heißt, READY wird bei jedem dieser Takte getestet, bis es aktiv ist und der Prozessor den Zyklus beenden kann. Auf diese Weise kann der Buszyklus an unterschiedliche Zugriffszeiten der Peripherie angepaßt werden.
Zu beachten ist in diesem Zusammenhang, daß im Falle eines Fehlers, z.B. durch einen defekten Speicher oder einen Programmfehler (Zugriff auf nicht vorhandene Adresse), der Prozessor in einen endlosen Wartezustand gerät. Deshalb sollte die Hardware eine Zeitüberwachung vorsehen, die einen solchen Zustand beendet ("Time Out").

2.2.2. Busstruktur

Zur Kommunikation mit dem Speicher und den Ein-/Ausgabekanälen ist eine Anzahl von Signalleitungen erforderlich. Die Bilder 2.6 und 2.7 zeigen zwei Möglichkeiten für die Implementierung eines Busses in einem System mit einem K1810WM86. Da die meisten Speicher- und Peripheriebausteine während des gesamten Buszyklus eine stabile Adresse verlangen, wird der Adreß-/Datenbus getrennt. Der Bus besteht aus folgenden Komponenten:

- Adreßbus (gespeichert),
- Datenbus (gepuffert oder ungepuffert),
- Steuerbus (Kommando- und Quittungssignale).

Für spezielle Anwendungen sind noch weitere Komponenten erforderlich:

- Interruptsignale (für interruptgesteuerte Ein-/Ausgabe),
- Busaustauschsignale (nur für Mehrrechnersysteme).

Bild 2.8 zeigt die prinzipielle Struktur eines Mehrrechnersystems. Bis zu 16 Mikroprozessoren können parallel in einem solchen System arbeiten. Ein gemeinsamer Systembus dient dem Austausch von Nachrichten. Der Zugriff auf diesen Systembus wird über besondere Steuerleitungen geregelt.

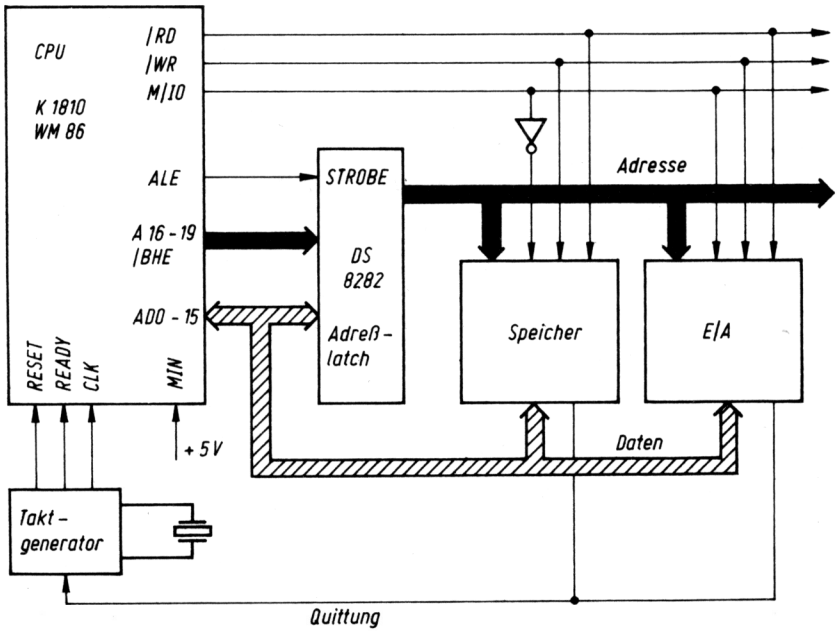


Bild 2.6. K1810WM86-System im Minimum-Mode

2.3. Speicherinterface

2.3.1. Speicheroperationen

Während einer Speicheroperation werden Daten zwischen einer Speicherzelle und dem Mikroprozessor transportiert. Der Speicher kann aus Schreib-/Lesespeichern (RAM) und Festwertspeichern (ROM bzw. EPROM) bestehen.
Der K1810WM86 kennt drei Arten von Speicheroperationen:

- Befehlskode lesen,
- Daten lesen,
- Daten schreiben.

Diese unterscheiden sich aus der Sicht der Hardware nur durch die Richtung des Datenflusses. Aus der Sicht der Software unterscheiden sie sich außerdem durch die Adressierung: Der Befehlskode wird stets mit Hilfe des CS-Registers adressiert, während sich Daten in den durch DS, ES oder SS definierten Segmenten befinden.

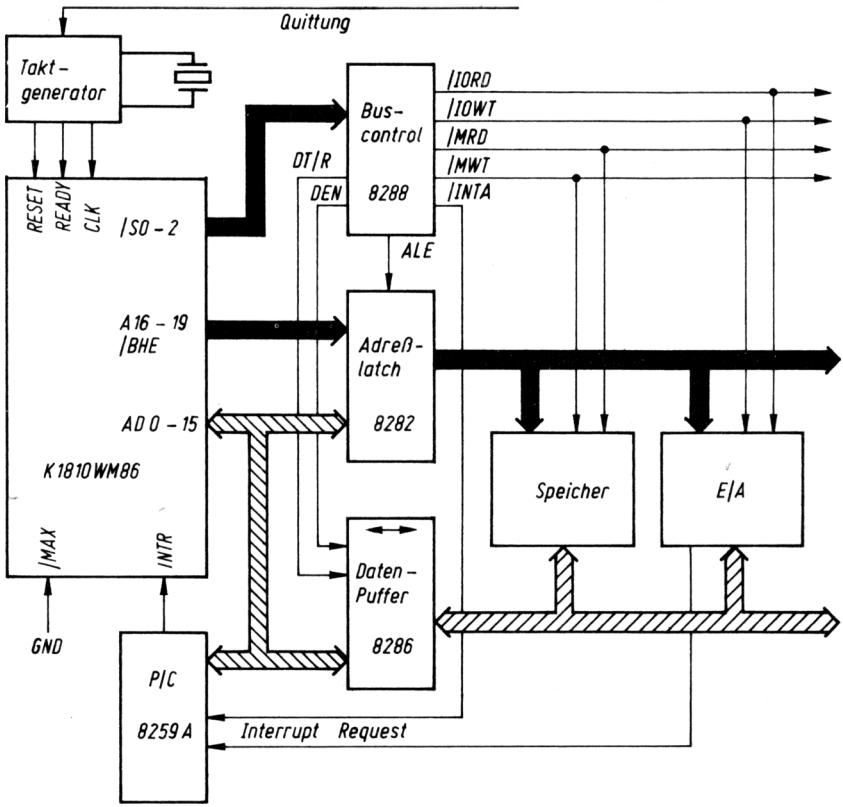


Bild 2.7. Einfaches K1810WM86-System im Maximum-Mode

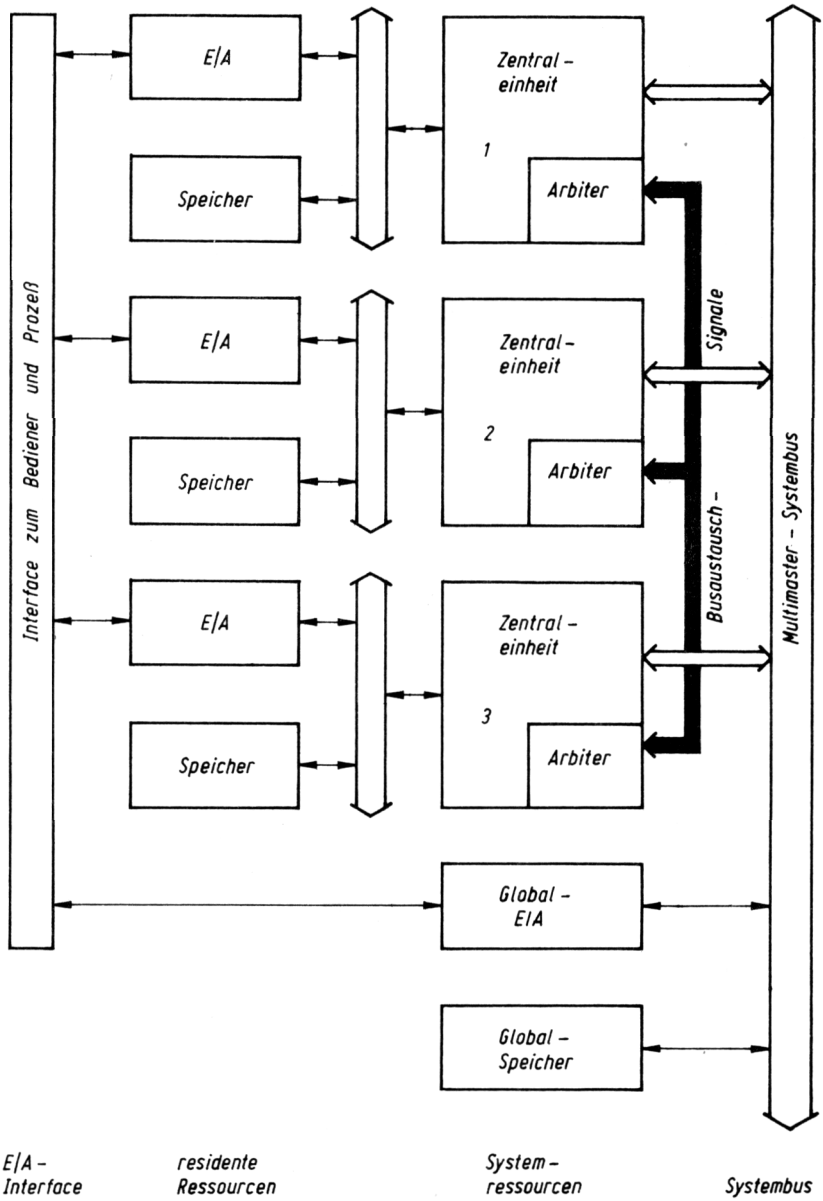


Bild 2.8. Prinzip eines Mehrrechnersystems

2.3.2. Speicherorganisation

Der K1810WM86 kann 1 MByte Speicher adressieren. Dieser ist physisch in zwei Blöcke zu je 512 KByte aufgeteilt. Ein Block ist an die untere, der andere an die obere Hälfte des Datenbusses angeschlossen. Das Bild 2.9 zeigt den Speicher aus der Sicht des Programmierers und den physischen Aufbau.
Ein bestimmter Speicherplatz in einer Bank wird mit den Adreßleitungen A1 bis A19 ausgewählt. Welche der beiden Bänke dabei angesprochen wird, entscheidet das Adreßbit A0 in Verbindung mit dem Signal /BHE. Bei allen geradzahligen Adressen, wenn also A0 = 0 ist, wird die an D0 bis D7 angeschlossene Bank adressiert. Diese bezeichnen wir deshalb als "even bank" (gerade, bezogen auf die Adresse) oder "low bank" (niederwertig, bezogen auf den Datenbus). Die andere Bank enthält dann zwangsläufig nur die Bytes mit ungeraden Adressen, deshalb die Bezeichnung "odd bank" (ungerade) bzw. "high bank" (höherwertig). Ihre Adressierung erfolgt durch L-Pegel auf der Leitung /BHE.

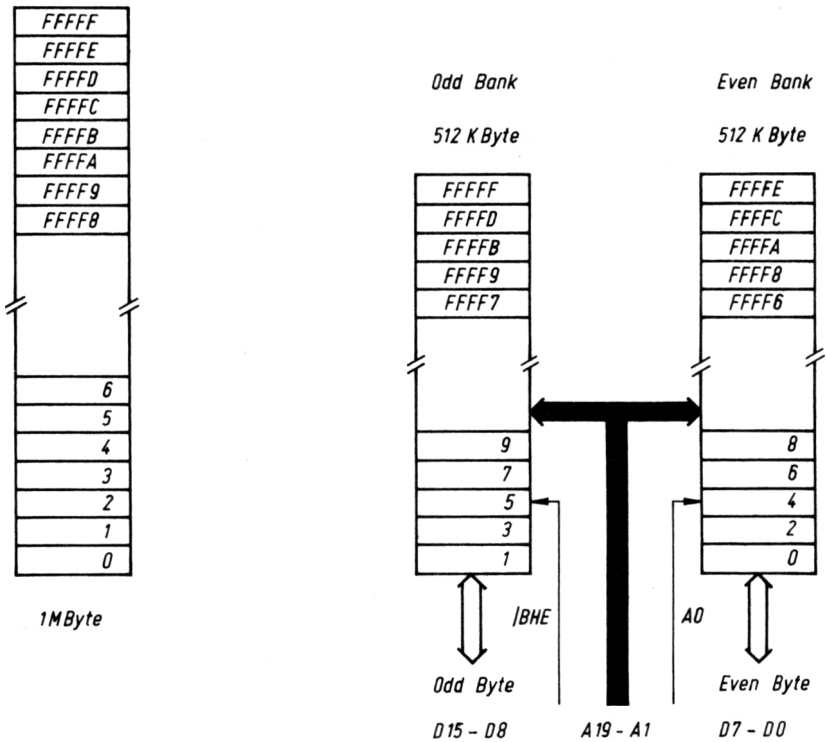


Bild 2.9. Speicherorganisation aus der Sicht des Programmierers (links) und physischer Speicheraufbau (rechts)

2.3.3. Byte- und Wortübertragungen

Der K1810WM86 kann Daten auf drei verschiedenen Wegen übertragen. Tafel 2.3 zeigt die drei Möglichkeiten.

Tafel 2.3. Datenbussteuerung

/BHE	A0	übertragene Daten
H	H	keine (unerlaubter Zustand)
H	L	Even-Byte (D0...D7)
L	H	Odd-Byte (D8...D15)
L	L	Wort (D0...D15)

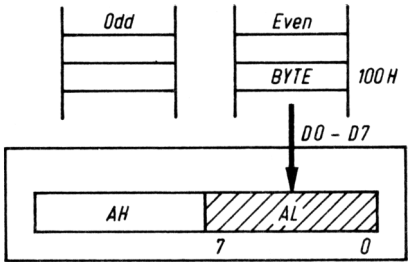
Wortübertragungen werden also nur bei geradzahligen Adressen ausgeführt, dabei werden gleichzeitig ein Even- und ein Odd-Byte von oder zu den Speicherplätzen der Adressen N und N+1 transportiert. Der Programmierer braucht diesen Vorgang nicht zu berücksichtigen, da der Prozessor Wortbefehle völlig unabhängig von der physischen Adresse akzeptiert. Der K1810WM86 überträgt Daten stets auf der richtigen Hälfte des Busses, und wenn ein Befehl einen Worttransfer mit einer ungeraden Adresse erfordert, so führt er automatisch zwei Byteübertragungen durch. Dabei wird zuerst das Odd-Byte transportiert, im nächsten Zyklus das Even-Byte von der nächsthöheren Adresse. Der einzige Effekt ist dabei die Verlängerung der Befehlsausführungszeit um einen Buszyklus.
Ein Beispiel soll die verschiedenen Möglichkeiten verdeutlichen:

Die Befehle

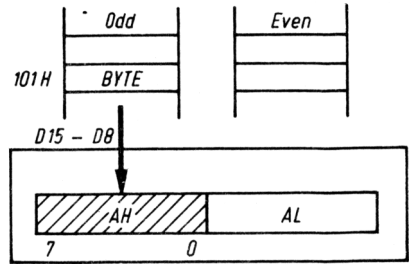
```
MOV AL,BYTE
MOV AH,BYTE
MOV AX,WORT
```

lesen den Inhalt der durch "BYTE" oder "WORT" vereinbarten Speicherzellen in den Akkumulator ein. Dabei ist es gleichgültig, ob die Speicheradresse gerade (z.B. 100H) oder ungerade (z.B. 101H) ist. Im Bild 2.10 sind die verschiedenen Operationen des K1810WM86 dargestellt.

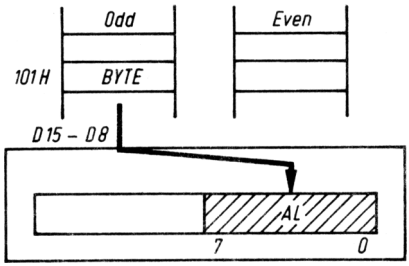
Beim Lesen von Befehlskodes ist es völlig gleichgültig, ob der Befehl auf einer geraden oder ungeraden Adresse beginnt. Befehlskodes werden durch Wortübertragungen eingelesen. Der Prozessor sortiert intern die Kodes byteweise in richtiger Reihenfolge in die Befehlswarteschlange ein.
Durch diese Arbeitsweise ist es möglich, daß der Programmierer den gesamten Arbeitsspeicher als byteweise organisiert betrachten kann und für Wortbefehle keine Einschränkungen bestehen. Damit werden eine effektive Speicherverwaltung und die Arbeit mit logischen Adressen begünstigt. Für eine laufzeitoptimale Programmierung können für Wortdaten gerade Adressen vereinbart werden.



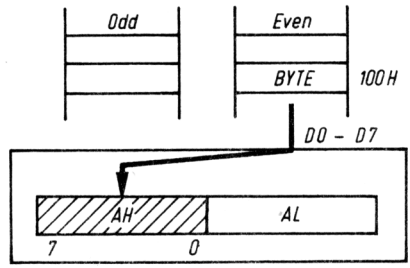
MOV AL, BYTE ; ADR 100H



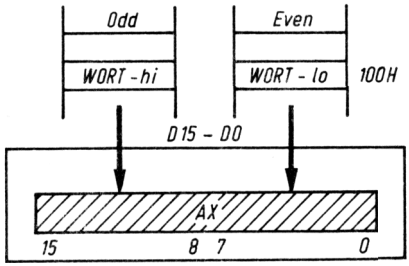
MOV AH, BYTE ; ADR 101H



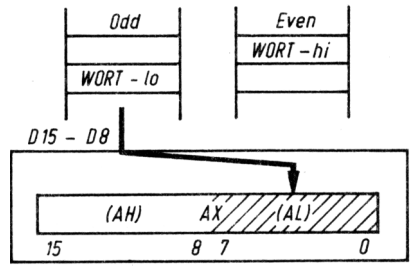
MOV AL, BYTE ; ADR 101H



MOV AH, BYTE ; ADR 100H

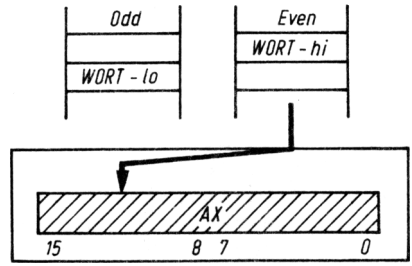


MOV AX, WORT ; ADR 100H



lo = LSB
(niederwertiges Byte)

hi = MSB
(höherwertiges Byte)



MOV AX, WORT ; ADR 101H

Bild 2.10. Byte- und Wortübertragungen

2.3.4. Segmentierung und physische Adressen

Während die Aufteilung in die beiden Bänke für die Programmierung keine Bedeutung hat, ist eine andere Art der Speicheraufteilung sehr wichtig - die Segmentierung.
Da der K1810WM86 nur 16-Bit-Register enthält, kann er nicht unmittelbar 1 MByte Speicher adressieren, denn dazu sind 20 Bit erforderlich. Deshalb wird der Speicherraum in logische Segmente unterteilt.
Ein Segment ist eine logische Einheit aufeinanderfolgender Speicherzellen, die maximal 64 KByte umfassen kann. Jedem Segment wird eine Basisadresse zugeordnet. Die Segmentregister enthalten die Zeiger auf die Basisadressen der vier Segmente, mit denen der Prozessor gleichzeitig arbeiten kann: Kodesegment, Stapelsegment und zwei Datensegmente. Innerhalb eines Segmentes werden die Speicherzellen durch einen Offset adressiert. Das ist eine 16-Bit-Speicheradresse, welche den Abstand von der Basisadresse angibt.
Jedes Programm kann entsprechend der zu lösenden Aufgabe mehrere Segmente definieren, die mit oder ohne Lücke aneinandergereiht sind oder sich teilweise oder ganz überlappen, wie Bild 2.11 zeigt.
Eine bestimmte Speicherzelle kann also verschiedenen Segmenten zugeordnet sein. Das heißt, sie hat zwar nur eine physische, aber mehrere logische Adressen. Die logische Adresse kann in der Form

BASIS : OFFSET

angegeben werden.

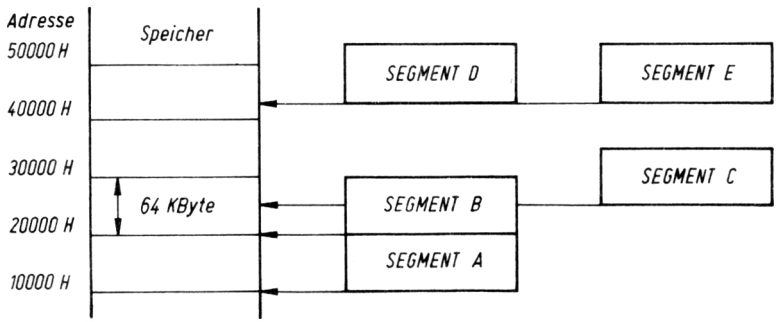


Bild 2.11. Logische Segmente im physischen Speicher

Die Bildung der physischen Adresse erfolgt auf folgende Weise:
Der Adreßgenerator in der BIU verschiebt den Inhalt des Segmentregisters um vier Bitpositionen nach links (Multiplikation mit 16). Dieser Wert entspricht dann der tatsächlichen Basisadresse des Segments. Dazu wird dann der jeweilige Offset addiert. Ein Beispiel soll dies verdeutlichen:

Das CS-Register enthält den Wert 13C0H. Die Offsetadresse des nächsten Befehls, welche im IP-Register steht, sei 0126H. Die physische Adresse, von der der Befehlskode gelesen wird, ist dann

```
13C0H x 16 =      13C00H
                +  0126H
                  13D26H
```

Die gleiche Zelle 21C26H kann von einem anderen Programm durch den Offset 06H adressiert werden, wenn das Segmentregister mit 13D2H geladen wurde. Die Adreßangaben 13C0:0126 und 13D2:0006 zeigen also beide auf die Zelle 13D26H (siehe auch Bild 2.12).

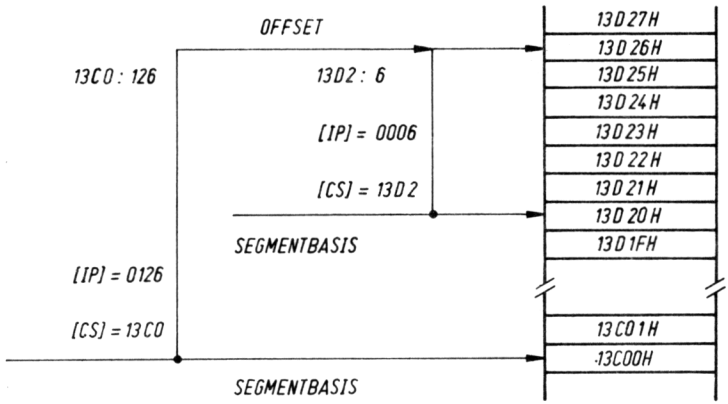


Bild 2.12. Logische und physische Adressen

Dieses Verfahren bedingt zum einen, daß alle Segmente auf Adressen beginnen, die ein Vielfaches von 16 sind. Außerdem wird dadurch erreicht, daß die Adressen innerhalb eines 64-KByte-Segments umlaufen. Das heißt, wenn durch einen Befehl die obere Segmentgrenze überschritten wird, springt der Adreßzähler wieder zum Anfang des Segments (auf den Offset FFFFH folgt 0H).

Die Segmentstruktur des K1810WM86- Speichers ermöglicht es, Programme zu schreiben, die unabhängig von ihrer Plazierung im Speicher sind. Sie können dynamisch verschoben werden, ohne daß man sie neu übersetzen muß (Dynamical Relocatable Code). So können beispielsweise in einem Multi-Task-System Programme, welche nicht benötigt werden, auf Diskette abgelegt werden, so daß der Speicher frei für andere Programme ist. Bei Bedarf können sie dann auf einen beliebigen freien Bereich geladen werden. Voraussetzung dafür ist, daß die Programme selbst die Segmentregister nicht beeinflussen.

2.3.5. Stapel

Ein besonderer Bereich des Speichers ist der Stapel (Stack). Dieser Bereich dient der Rettung wichtiger Programmdaten und der Registerinhalte (einschließlich der Flags) bei Programmverzweigungen, welche durch den Aufruf von Unterprogrammen oder bei der Reaktion auf Interruptanforderungen entstehen. Oft wird der Stapel auch für die Übergabe von Parametern benutzt, die ein Unterprogramm benötigt. Er funktioniert nach dem LIFO-Prinzip, das bedeutet "zuletzt hinein, zuerst heraus (last in, first out)".

Beim K1810WM86 wird der Stapel vom aktuellen Programm als besonderes Speichersegment kreiert. Die Anzahl der maximal 64 KByte großen Stapelsegmente ist dadurch nicht begrenzt.
Die Basisadresse ist durch das Stapelsegmentregister SS gegeben, der Offset durch den Stapelzeiger SP. Der Offset gibt den Abstand der letzten abgelegten Information in Byte von der Basisadresse an. Alle Stapeloperationen (PUSH, POP) sind Wortübertragungen.
Nach der Initialisierung von SS ist SP zunächst auf 0000H gesetzt. Beim Ablegen einer Information (PUSH) wird zunächst SP um den Wert 2 erniedrigt und das entsprechende Datenwort zu diesem Speicherplatz transportiert. Der Offset für das erste abgelegte Wort ist also FFFEH! Der Stapel wird demzufolge von seinem oberen Ende beginnend gefüllt und wächst in Richtung seiner Basis. Entsprechend wird beim Auslesen aus dem Stapel (POP) zunächst das Datenwort gelesen und dann SP um den Wert 2 erhöht. Ein Überlauf bewirkt das Überschreiben des Stapelanfangs.
In der Praxis wird meist ein Stapelbereich vereinbart, der kleiner als 64 KByte ist. Zu diesem Zweck muß außer SS auch SP initialisiert werden, indem es mit der gewünschten Endadresse geladen wird. Bei Stapelüberlauf ist zu beachten, daß dann die Daten außerhalb des vereinbarten Bereiches abgelegt werden.

Beim Aufruf von Interruptprozeduren wird stets automatisch PSW, CS und IP in den Stapel übertragen und bei der Rückkehr in das Hauptprogramm in umgekehrter Reihenfolge ausgelesen. Ähnliches gilt auch beim Aufruf einer Prozedur durch einen CALL-Befehl. Liegt die aufgerufene Prozedur innerhalb des aktuellen Kodesegments (Intrasegment), wird nur IP gerettet. Liegt sie dagegen außerhalb (Intersegment), werden CS und IP gerettet. Werden zu Beginn des Unterprogramms andere Daten über PUSH-Befehle abgelegt, so müssen diese vor der Rückkehr in umgekehrter Reihenfolge wiedergeholt werden (POP).

Darüber hinaus kann das Stapelsegment wie ein allgemeines Datensegment behandelt werden, das heißt, die Daten im Stapel können als Operanden für die meisten der K1810WM86- Befehle benutzt werden. Zur Adressierung von Daten im Stapel, beispielsweise zu übergebende Parameter, kann das BP-Register benutzt werden. Diese Möglichkeit wird oft bei der Unterprogrammbehandlung durch Compiler für höhere Programmiersprachen genutzt.

2.4. Ein-/Ausgabeinterface

Eingabeoperationen transportieren Daten von einem Eingabeport zum Prozessor, Ausgabeoperationen vom Prozessor zu einem Ausgabeport. Ein-/Ausgabekanäle werden zum Anschluß von Geräten zur Mensch-Maschine-Kommunikation benutzt (Bildschirm, Tastatur, Platten- oder Folienspeicher u.ä.). Außerdem existieren in jedem Computer weitere, nach außen nicht unbedingt sichtbare Ein-/Ausgabekanäle, zum Beispiel Zähler und Zeitgeber. Nicht vergessen darf man schließlich die Standardinterfaces, die dem universellen Geräteanschluß und der Kommunikation mit anderen Computern dienen, zum Beispiel das serielle V.24-Interface.
Der K1810WM86 verfügt nur über eine relativ bescheidene Systemunterstützung hinsichtlich der Ein-/Ausgabe. Der Adreßraum umfaßt 64 KByte, das sind 65536 Ports. Die Portadresse muß normalerweise im DX-Register enthalten sein; für die ersten 256 Ports ist auch eine direkte Adressierung über den Befehlskode möglich (Adressen 0H bis 0FFH). Ziel oder Quelle der Operation ist immer der Akkumulator.
Die Buszyklen unterscheiden sich nicht von Speicherzyklen, lediglich das Signal M/IO bzw. /S2 definiert den Ein-/Ausgabeadreßraum. Das bedeutet, daß Byte- und Wortübertragungen möglich sind und nach denselben Regeln wie Speicheroperationen durchgeführt werden und Ein-/Ausgabeports sowohl an das Low- als auch an das High-Byte des Datenbusses angeschlossen sein können. In der Regel sind Ein-/Ausgabekanäle byteweise organisiert und an die untere Hälfte des Datenbusses angeschlossen. Sie werden dann nur mit geraden Adressen angewählt.

Speicheradressierte Ein-/Ausgabe:
Für ein-/ausgabe-intensive Systeme ist es manchmal vorteilhaft, die Portadressen als Teil des Speicherraumes zu betrachten, indem im physischen Speicher ein Bereich dafür reserviert wird. Dieser Bereich wird als Datensegment behandelt. Dadurch ist der gesamte Befehlssatz, beispielsweise Blockübertragungen oder Bitmanipulationen, auf die Ein-/Ausgabekanäle anwendbar.

2.5. Interruptbehandlung

2.5.1. Übersicht

Für die Leistungsfähigkeit eines Mikroprozessors ist dessen Vermögen, auf Bedienungsanforderungen von peripheren Geräten oder beim Auftreten besonderer Ereignisse zu reagieren, von besonderer Bedeutung. Solche Programmunterbrechungen (Interrupts) können auftreten, wenn beispielsweise Daten von einem externen Interface abgeholt werden müssen oder eine Bedienhandlung vorgenommen wurde; aber auch, wenn ein Netzspannungsausfall auftritt. Diese Ereignisse sind mit bestimmten Prioritäten behaftet; der Prozessor muß dementsprechend schnell reagieren, bestimmte Anforderungen zurückstellen oder verbieten können.

Die Anerkennung einer Interruptanforderung ist stets mit einer Programmverzweigung verbunden, bei der das aktuelle Programm verlassen und in ein anderes Programm, eine Interruptprozedur, gesprungen wird.
Der K1810WM86 verfügt über ein einfaches, aber leistungsfähiges Interrruptsystem. Jedem Interrupt ist eine Typnummer als Erkennungskode zugeordnet. Es können 256 verschiedene Typen definiert werden. Die Auslösung einer Interruptanforderung kann auf drei verschiedenen Wegen erfolgen:

- Hardwaregesteuerte Interrupts: Signale INTR, NMI,
- Programmgesteuerte Interrupts: Befehle INTO, INTn,
- Prozessorinterne Interrupts: Divisionsfehler, Einzelschritt.

Im Bild 2.13 sind diese Möglichkeiten dargestellt.

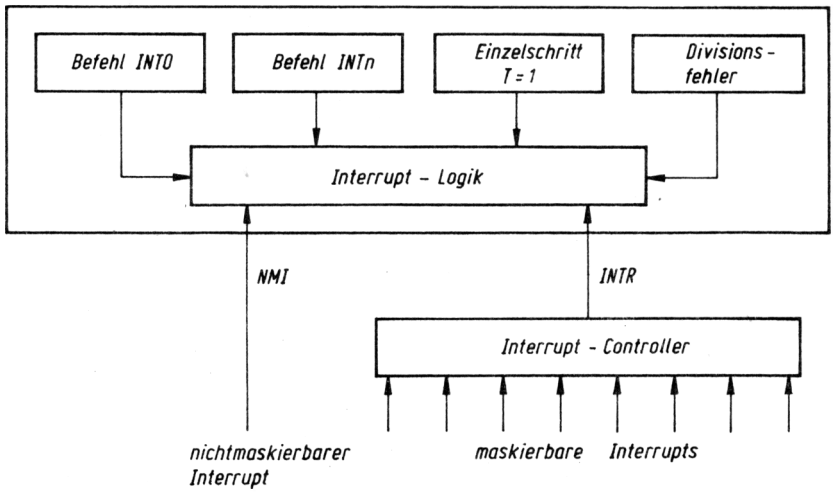


Bild 2.13. Interruptquellen

2.5.2. Interruptvektortabelle

Die Vektortabelle ist das Bindeglied zwischen dem Interrupt und der zugehörigen Prozedur. Die Vektortabelle belegt stets die Adressen 00000H bis 003FFH, das ist das erste Kilobyte des Speichers. Sie enthält maximal 256 Doppelwortzeiger, das heißt, jeder Interrupttyp belegt 4 Byte. Diese enthalten das Kodesegment (CS) und den Offset (IP) der Startadresse der Interruptprozedur. Dadurch kann diese an jeder beliebigen Stelle im Speicher angeordnet sein. Bild 2.14 zeigt den Aufbau der Interruptvektortabelle.

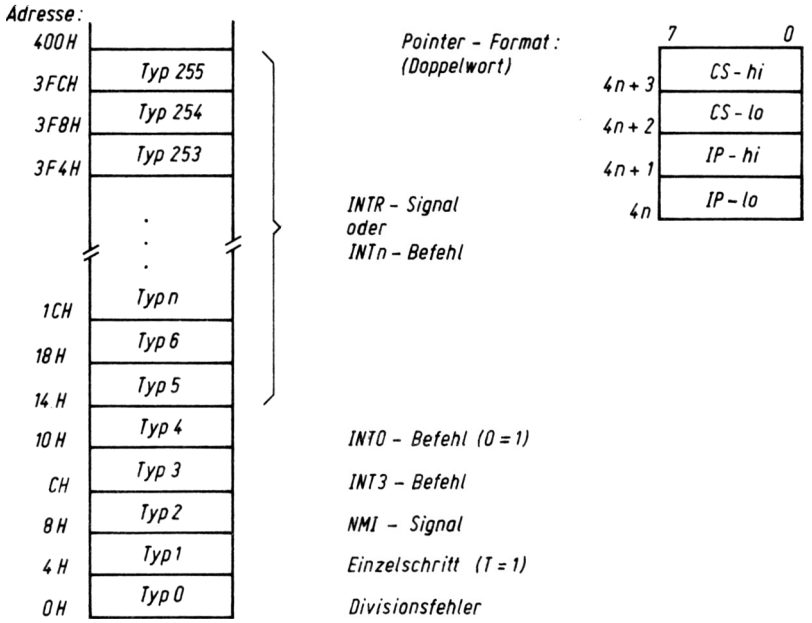


Bild 2.14. Aufbau der Interruptvektortabelle

2.5.3. Interrupts mit fest zugeordnetem Typ

Die ersten fünf Zeiger der Tabelle sind bestimmten Interrupttypen fest zugeordnet:

TYP 0 : Divisionsfehler
Dieser Interrupt ist Bestandteil der Divisionslogik. Er ist nicht sperrbar und hat höchste Priorität.
Überschreitet der Quotient einer Division den zulässigen Wert (Überlauf von AX bzw. DX), so wird die durch den Vektor 0 adressierte Prozedur aufgerufen.

TYP 1 : Einzelschrittbetrieb (Single Step)
Das Testflag T im PSW dient dem Testen von Befehlsfolgen im Einzelschrittbetrieb. T kann durch keinen Befehl direkt, sondern nur auf folgende Weise gesetzt werden: Zuerst muß PSW in den Stapel übertragen werden, dann kann dort Bit 8 gesetzt werden, und anschließend muß PSW wieder geladen werden. Nach der Abarbeitung des nächsten Befehls wird in das durch Vektor 1 adressierte Unterprogramm gesprungen; dabei wird automatisch PSW wieder in den Stapel übertragen. Der Prozessor setzt intern T auf 0, so daß das Unterprogramm normal abgearbeitet wird. Bei der Rückkehr in das zu testende Hauptprogramm wird PSW aus dem Stapel rekonstruiert; T ist automatisch wieder gesetzt. Es wird ein weiterer Befehl abgearbeitet und der ganze Vorgang wiederholt sich so oft, bis T im Stapel rückgesetzt wird.

Auf diese Weise können Programme Befehl für Befehl getestet werden, indem das Unterprogramm beispielsweise Registerinhalte oder andere interessierende Zustände anzeigt.

TYP 2 : Nichtmaskierbarer Interrupt (NMI)
Die durch Vektor 2 adressierte Prozedur wird aufgerufen, wenn das Signal NMI aktiv wird. Sie ist im Gegensatz zu anderen externen Interruptanforderungen nicht sperrbar (maskierbar) und besonderen, hochpriorisierten Hardwarequellen vorbehalten, etwa einem bevorstehenden Spannungsausfall oder einer Echtzeituhr.

TYP 3 : Befehl INT 3
Wird der Einbytebefehl INT 3 ausgeführt, springt das Programm in die durch Vektor 3 adressierte Prozedur. Ähnlich wie der Einzelschrittmodus ist dieser Interrupt als Testhilfe gedacht und kann zur Programmierung von Haltepunkten ("Break Points") benutzt werden. Soll ein zu testendes Programm an einer bestimmten Stelle gestoppt werden, um bestimmte Systemzustände zu testen, so wird das nächste Befehlsbyte durch den Kode für INT 3 ersetzt. Die Interruptprozedur kann dann die entsprechenden Tests durchführen und auswerten.

TYP 4 : Befehl INTO (Interrupt On Overflow)
Mit diesem Befehl kann ein bedingter Sprung in die durch Vektor 4 adressierte Prozedur programmiert werden. Bedingung ist, daß das Überlaufflag O auf 1 gesetzt ist.

Die restlichen 251 Interruptvektoren können entweder durch Software-Interrupts mit dem Befehl INT n oder durch externe maskierbare Interruptanforderungen über das Signal INTR aufgerufen werden.

2.5.4. Externe Interruptanforderungen

Die externen Interruptanforderungen sind asynchrone, durch die Peripherie ausgelöste Ereignisse, auf die der Mikroprozessor reagieren soll. Der K1810WM86 kann zwei externe Interruptanforderungen bearbeiten. Dafür verfügt er über zwei Signaleingänge:

INTR (Interrupt Request)
Anforderungen über diese Leitung sind maskierbar, das heißt, durch das Programm sperrbar. Dazu kann das Interruptflag I durch die Befehle STI und CLI beeinflußt werden.
Das Signal INTR wird am Ende eines jeden Befehls getestet. Ist es aktiv, jedoch I = 0, so ignoriert der Prozessor die Anforderung und führt den nächsten Befehl aus. (Da das Signal nicht abgespeichert wird, muß es bis zur Anerkennung aktiv gehalten werden, sonst geht es verloren.) Ist I = 1, so wird der nächste Befehl nicht abgearbeitet, sondern eine Interruptanerkennung eingeleitet, um den Interrupttyp zu ermitteln.
Beim Sprung in die Interruptprozedur wird automatisch I zu-

rückgesetzt, so daß weitere maskierbare Interrupts verboten sind. Nach der Rekonstruktion der Register aus dem Stapel bei der Rückkehr, das heißt nach dem Befehl IRET, ist I wieder gesetzt. Soll die Prozedur unterbrechbar sein, muß in ihr das Setzen von I programmiert sein. Nach dem Setzen dieses Flags wird stets noch ein Befehl ausgeführt, erst dann können Anforderungen wieder akzeptiert werden.
Nach RESET ist I automatisch zurückgesetzt, so daß erst nach einem STI-Befehl maskierbare Interrupts wirksam werden können.

NMI (Non Maskable Interrupt)
Eine Anforderung über diese Leitung ist nicht sperrbar. Das Signal ist flankengetriggert und wird vom Prozessor abgespeichert. (Es soll eine Dauer von wenigstens 2 Taktperioden haben, jedoch können auch wesentlich kürzere Impulse zur Auslösung führen; deshalb ist auf eine gute Entstörung zu achten.) Am Ende eines jeden Befehls wird der NMI-Speicher abgefragt. Ist sowohl NMI als auch INTR aktiv, so hat NMI die höhere Priorität. Es werden keine Anerkennungszyklen ausgeführt, sondern es wird sofort ein Sprung in die vom Interruptvektor 2 adressierte Prozedur ausgeführt.

Es gibt einige Befehle, bei denen hinsichtlich der Interruptanerkennung Besonderheiten zu beachten sind:

- Alle Präfixe (Lock, Repeat, Segment Override) werden als Bestandteil des folgenden Befehls interpretiert und sind deshalb nicht unterbrechbar.
- Nach allen Befehlen, welche Daten in ein Segmentregister laden, wird erst noch ein zweiter Befehl ausgeführt. Das bietet die Möglichkeit, daß noch ein zugehöriges Zeigerregister geladen werden kann.
- Bei Stringoperationen mit Repeat-Präfix werden Interruptanforderungen nach jeder Einzeloperation anerkannt. Dabei geht jedoch ein eventueller zweiter Präfix verloren.

2.5.5. Interne Interrupts

Alle Interruptbefehle, das gesetzte Testflag und Divisionsfehler führen zu internen Interrupts. Im Gegensatz zu den externen Interrupts werden sie durch die Programmabarbeitung ausgelöst und treten insofern nicht asynchron auf. Sie sind durch folgende Merkmale gekennzeichnet:

- Der Typ steht entweder fest (Typ 0, 1, 3, 4) oder ist im Befehlskode enthalten. Es werden keine Anerkennungszyklen ausgeführt.
- Sie sind nicht maskierbar und werden sofort ausgeführt.
- Sie besitzen höhere Priorität als externe Anforderungen (mit Ausnahme des Einzelschrittmodus).

2.5.6. Interruptanerkennung

Die Anerkennung einer Interruptanforderung bewirkt die folgenden Operationen, die der K1810WM86 automatisch ausführt:

1. Bei maskierbaren Interruptanforderungen werden zwei Buszyklen generiert, die dem Lesen des Vektorbytes von einem externen Interruptcontroller-Baustein dienen. Bei allen anderen Anforderungen werden diese Anerkennungszyklen nicht durchgeführt. Diese Buszyklen entsprechen Lesezyklen, anstelle des /RD-Signals wird aber das Signal /INTA aktiviert. Im ersten Zyklus wird der Systemzustand eingefroren und den externen Interruptcontroller- Bausteinen die Möglichkeit gegeben, den Typ der aktuellen höchstpriorisierten Anforderung zu ermitteln. Im zweiten Zyklus liest der Prozessor ein vom Interruptcontroller bereitgestelltes Datenbyte ein, welches als Typnummer für die Interruptvektortabelle interpretiert wird. Der K1810WM86 multipliziert es mit vier und erhält die Adresse, auf der in der Vektortabelle der Zeiger auf die zugehörige Prozedur eingetragen ist.

Bei allen Interruptanforderungen werden anschließend die folgenden Operationen durchgeführt:

2. Der K1810WM86 liest die Werte für den Offset (IP) und das Kodesegment (CS), das heißt die Startadresse der Interruptprozedur aus der Vektortabelle.
3. Der Inhalt des PSW wird in den Stapel übertragen.
4. Die Flags I und T werden zurückgesetzt.
5. Die bisherigen Inhalte von CS und IP werden in den Stapel übertragen.
6. CS und IP werden mit den neuen Werten geladen und der erste Befehl der Interruptprozedur wird eingelesen und ausgeführt.

Die Prozedur sollte alle Register und Parameter, die von ihr verändert, jedoch nach der Rückkehr wieder benötigt werden, in den Stapel ablegen und am Ende der Prozedur wieder auslesen. Den Abschluß der Interruptprozedur bildet der Befehl IRET. Dieser bewirkt, daß IP, CS und PSW wieder aus dem Stapel geladen werden und das Programm, welches unterbrochen wurde, an der richtigen Stelle fortgesetzt wird. Der gesamte Ablauf ist im Bild 2.15 dargestellt.

Die Tafel 2.4 enthält die Ausführungszeiten für die einzelnen Interruptanforderungen, das heißt die Zeit von der Erkennung bis zum Beginn des ersten Befehls der Interruptprozedur. Zu diesen Zeiten können noch Wartetakte bei den Buszyklen und die Latenz vom Auftreten bis zur Erkennung der Anforderung kommen. Letztere ist durch die Zeit bestimmt, welche der Prozessor zur Abarbeitung des aktuellen Befehls benötigt.

Tafel 2.4. Charakteristik der Interruptklassen

Priorität	Klasse	Ausführungszeit
höchste	Divisionsfehler	50 Takte
↑	INT 3 INTO INT n	52 Takte 53 Takte 51 Takte
	NMI	50 Takte
	INTR	61 Takte
niedrigste	Einzelschritt	50 Takte

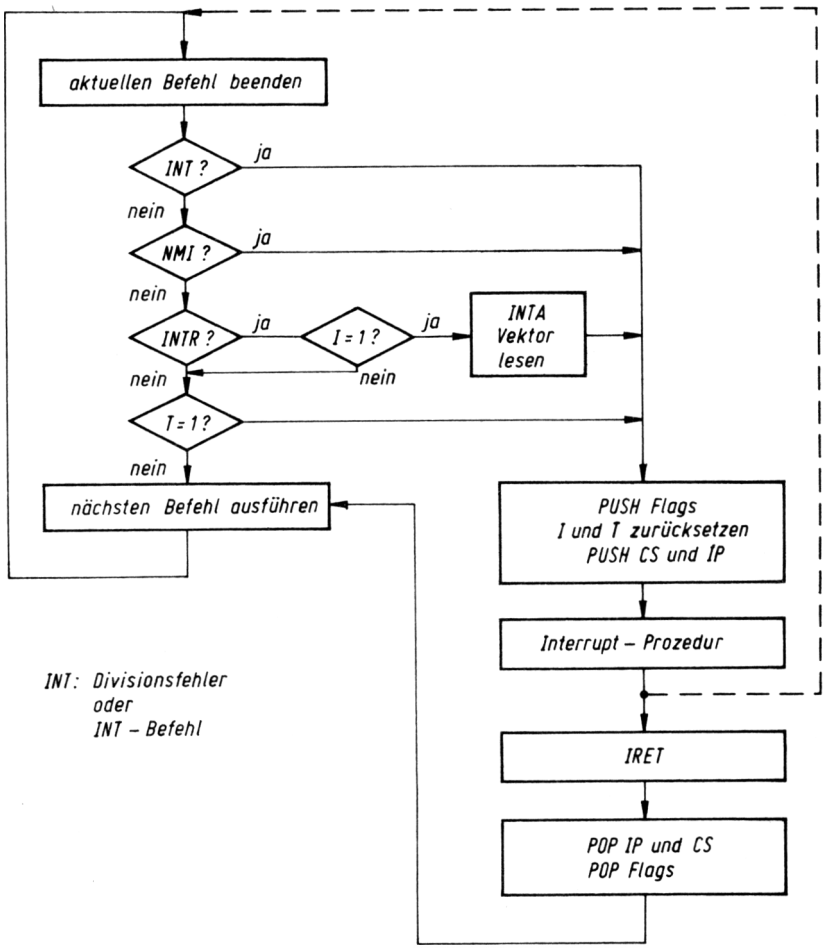


Bild 2.15. Ablauf der Interruptbehandlung

2.6. Prozessorsteuerung

2.6.1. Reset

Der RESET-Eingang des K1810WM86 dient dem Starten des Systems, zum Beispiel nach dem Zuschalten der Betriebsspannung. Solange RESET H-Potential führt, ist der Prozessor inaktiv. (Alle TRI-STATE-Signale sind hochohmig.)
Geht RESET auf L-Potential, nimmt der Prozessor folgenden Zustand ein:

- Alle Flags sind auf 0 gesetzt.
- IP ist mit 0000H geladen.
- CS ist mit FFFFH geladen.
- DS, ES und SS sind mit 0000H geladen.
- Die Befehlswarteschlange (Instruction Queue) ist leer.

Anschließend springt der Prozessor auf die Startadresse FFFF0H und liest einen Befehl. Dieser ist üblicherweise ein Sprung auf den Beginn des Programms, beispielsweise die Initialisierung.
Der Adreßbereich von FFFF0H bis FFFFFH ist für das System-RESET reserviert.

2.6.2. Busanforderungen

Der K1810WM86 verfügt über einen Mechanismus, mit dessen Hilfe der Bus mit einem anderen Prozessor oder DMA-Controller-Baustein (Direct Memory Access) geteilt werden kann.

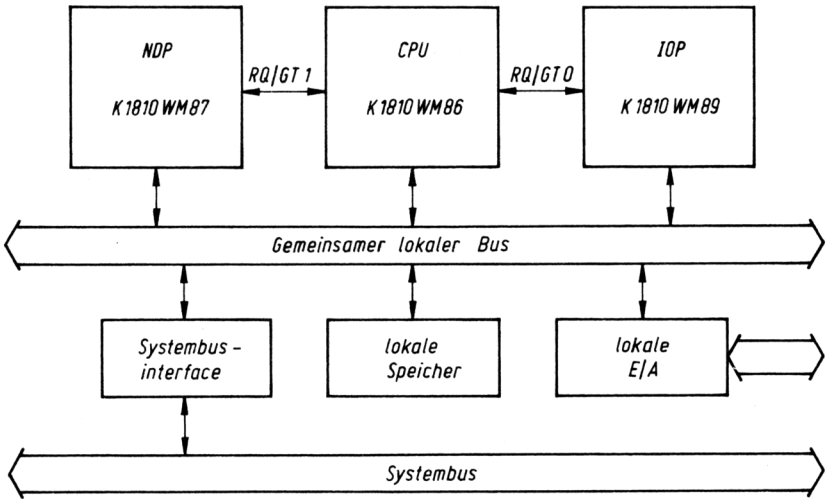


Bild 2.16. Coprozessoren am gemeinsamen Bus

Im Minimum-Mode gestatten die Leitungen HOLD und HOLDA die Zusammenarbeit mit DMA-Bausteinen, die einen schnellen Datentransfer zwischen Ein-/Ausgabegeräten und dem Speicher ermöglichen. Der DMA-Baustein fordert zu diesem Zweck durch Aktivieren der Leitung HOLD den K1810WM86 auf, ihm die Buskontrolle zu übertragen. Dieser beendet seinen aktuellen Buszyklus und signalisiert über die Leitung HOLDA, daß er den inaktiven (hochohmigen) Zustand eingenommen hat. Wird HOLD inaktiv, so übernimmt der K1810WM86 wieder die Bussteuerung.

Im Maximum-Mode können mehrere Coprozessoren mit dem K1810WM86 zusammenarbeiten, zum Beispiel der Numerikcoprozessor K1810WM87 und der Ein-/Ausgabeprozessor K1810WM89. Die Bussteuerung erfolgt über die bidirektionalen Signalleitungen RQ/GT0 und RQ/GT1, an die jeweils ein Coprozessor angeschlossen werden kann. Ein Impuls vom Coprozessor signalisiert die Busanforderung, ein Impuls vom K1810WM86 die Übergabe der Buskontrolle. Hat der Coprozessor seine Aktivität beendet, signalisiert er das durch einen weiteren Impuls. Anforderungen auf RQ/GT0 sind höherpriorisiert als solche auf RQ/GT1. Bild 2.16 zeigt den Aufbau eines Systems mit Coprozessoren.

Hat die BIU den Bus abgegeben, arbeitet die EU intern weiter, bis ein Buszugriff erforderlich ist.
Busanforderungen werden vom K1810WM86 unter folgenden Bedingungen akzeptiert:

- Die BIU hat einen Buszyklus beendet (T4-Takt).
- Die BIU ist passiv (TI-Takte).
- Der Prozessor ist im HALT-Zustand.

Busanforderungen werden nicht akzeptiert, wenn ein Befehl durch den Lock-Präfix geschützt ist (auch im Minimum-Mode!).
Eine Busanforderung hat gegenüber anstehenden Interruptanforderungen stets höhere Priorität. Sie wird auch unmittelbar nach RESET (also vor dem ersten Befehl) akzeptiert.

2.6.3. Busblockierung

In Mehrrechnersystemen, in denen mehrere Prozessoren auf gemeinsame Ressourcen zugreifen, ist es erforderlich, bestimmte Operationen zu schützen. Damit soll verhindert werden, daß Speicherdaten, die ein Prozessor benötigt, durch einen anderen zerstört werden.

Ein Beispiel hierfür sind "Semaphore", Speicherzellen, welche den Zustand eines zum Nachrichtenaustausch vereinbarten gemeinsamen Speicherbereichs repräsentieren (z.B. "leer" oder "besetzt"). Wenn zwischen Lesen und Schreiben (Setzen) ein anderer Prozessor Zugriff auf das Semaphor hat, kann dessen Inhalt verfälscht werden (siehe auch Abschn. 4.2.1, Befehl XCHG).

Das Signal /LOCK kann benutzt werden, um der Hardware, welche die Zuteilung des Zugriffs auf gemeinsame Ressourcen organisiert, eine Blockierung des Zugriffs anzuzeigen. Solange dieses Signal aktiv ist, werden alle Zugriffswünsche anderer Prozessoren unabhängig von ihrer Priorität ignoriert.
Das Signal /LOCK wird aktiviert, wenn ein Befehl mit dem Lock-Präfix versehen wird (z.B. LOCK XCHG). Es werden stets nur die Buszyklen eines Befehls geschützt. Zwischen zwei mit dem Lock-Präfix versehenen Befehlen wird das Signal inaktiv.
Befehle mit Lock-Präfix sind außerdem vor Busanforderungen geschützt. Damit wird gesichert, daß bestimmte Operationen schnellstmöglich ausgeführt werden.
Während der beiden Interruptanerkennungszyklen wird das Signal /LOCK automatisch aktiviert.

2.6.4. Wait und Test

Wird ein WAIT-Befehl abgearbeitet, so testet der K1810WM86 den Zustand des Eingangs /TEST. Führt dieses Signal L-Potential, führt der Prozessor den nächsten Befehl aus. Ist der Eingang nicht aktiv, so nimmt der Prozessor den passiven Zustand ein. Im Abstand von 5 Takten wird der /TEST-Eingang abgefragt, bis er aktiv ist. Dadurch kann das Programm mit externen Ereignissen synchronisiert werden. Ein Beispiel hierfür sind "Unterprogramme", welche durch einen ESCAPE-Befehl in einem Coprozessor gestartet werden:

Der K1810WM86 überträgt eine Aufgabe an den Coprozessor und arbeitet anschließend weiter, bis ein WAIT-Befehl signalisiert, daß er das Ergebnis des Coprozessors benötigt. Dieser aktiviert die Leitung /TEST, wenn er seine Aufgabe gelöst hat.

Im WAIT-Zustand akzeptiert der K1810WM86 externe Interrupts. Die Interruptprozedur wird unabhängig vom Zustand der /TEST-Leitung abgearbeitet. Der K1810WM86 kehrt aber wieder zum WAIT-Befehl zurück, so daß der WAIT-Zustand wieder eingenommen wird, falls der /TEST-Eingang noch inaktiv ist.

2.6.5. Halt-Zustand

Neben dem passiven Zustand, der eingenommen wird, wenn die BIU keine Daten zu übertragen hat und auch die Befehlswarteschlange gefüllt ist, gibt es noch einen weiteren Zustand, in dem der Prozessor nach außen keine Aktivitäten zeigt. Diesen inaktiven Zustand nimmt der Prozessor nach der Abarbeitung eines HALT-Befehls ein. Dabei führen weder EU noch BIU irgendwelche Operationen durch außer dem Testen der Leitungen INTR und NMI. Erst eine Interruptanforderung auf einer dieser Leitungen oder RESET befreien den Prozessor aus diesem Zustand, und das Programm wird mit der Interruptroutine oder einer Neuinitialisierung fortgesetzt.

3. Adressierungsarten

Dieser Abschnitt beschreibt, wie der K1810WM86 auf Operanden zugreift. Die Adressierungsarten sind die Voraussetzung für die effektive Befehlsstruktur des K1810WM86.

3.1. Übersicht

Die meisten der K1810WM86-Befehle enthalten zwei Operanden: einen Ziel- und einen Quelloperand. Der K1810WM86 verfügt über zahlreiche Möglichkeiten, die Operanden eines Befehls zu adressieren. Bei einigen Befehlen ist einer oder beide Operanden implizit vorgegeben, das heißt, sie werden in der Anweisung nicht mitgeschrieben. Bei den meisten Befehlen müssen jedoch Operanden angegeben werden, da sie den Befehlstyp bestimmen. Für Speicher und Ein-/Ausgabeports muß der K1810WM86 physische Adressen berechnen, die er in den Buszyklen aussendet.

Beispielsweise überträgt der Befehl

```
MOV AX,ALPHA
```

ein Wort von einer Speicherzelle "ALPHA" zum AX-Register. ALPHA ist der Quelloperand. Der Prozessor muß die Speicheradresse bestimmen. Die Vorschrift dazu enthält der Befehlskode. AX ist der Zieloperand, der ebenfalls im Befehlskode spezifiziert wird. Verwendet der MOV-Befehl andere Operanden, so muß zwangsläufig ein anderer Befehlskode entstehen.

Der K1810WM86 kann vier verschiedene Operandentypen adressieren:

- Register,
- unmittelbare Werte (nur als Quelloperand),
- Speicherdaten,
- Ein-/Ausgabeports.

Zur Bildung der Speicheradressen können gleichzeitig vier Elemente benutzt werden:

- Segmentregister,
- Basisregister,
- Indexregister,
- Displacement (konstanter Abstand).

Je nachdem, welche der Elemente vorhanden sind, kann der K1810WM86 fünf verschiedene Adressierungsarten anwenden:

- direkt,
- indirekt,
- relativ,
- indiziert,
- basisindiziert.

Bild 3.1 zeigt schematisch die Berechnung der physischen Speicheradresse aus diesen Elementen.
Die Tafel 3.1 zeigt einige Möglichkeiten der Darstellung verschiedener Adressierungsarten in Assemblersprache.

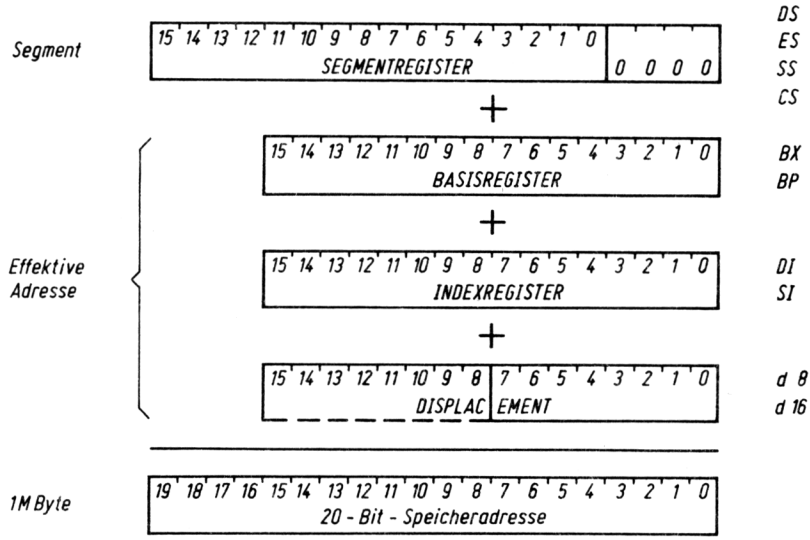


Bild 3.1. Prinzip der Speicheradressierung

Tafel 3.1. Beispiele für Adressierungsarten

Kode-Beispiel	Zieloperand , Quelloperand
MOV AX,BX	; REGISTER , REGISTER
MOV CL,5	; REGISTER , WERT
MOV BX,ALPHA	; REGISTER , SPEICHER (DIREKT)
MOV AL,[BX]	; REGISTER , SPEICHER (INDIREKT)
MOV DI,BETA [BP]	; REGISTER , SPEICHER (RELATIV)
MOV DX,GAMMA [SI]	; REGISTER , SPEICHER (INDIZIERT)
MOV BP,DELTA [BX+SI]	; REGISTER , SPEICHER (BASISINDIZIERT)
ADD ALPHA+5,2H	; SPEICHER (DIREKT) , WERT
ADD [DI],AL	; SPEICHER (INDIREKT) , REGISTER
ADD BETA [BP],CX	; SPEICHER (RELATIV) , REGISTER
ADD GAMMA [DI],BX	; SPEICHER (INDIZIERT) , REGISTER
ADD [BP+DI],SI	; SPEICHER (BASISINDIZIERT) , REGISTER
OUT OC2H,AL	; PORT (DIREKT) , AKKUMULATOR
IN AX,DX	; AKKUMULATOR , PORT (INDIREKT)

3.2. Registeroperanden

Der Operand ist ein Registerinhalt. Für den Zugriff auf die Daten sind keine Buszyklen erforderlich. In welchem Register der oder die Operanden stehen, wird im Operationskode spezifiziert, so daß diese Befehle kompakt sind (ein oder zwei Byte) und kurze Ausführungszeiten haben. Einige Befehle haben feststehende Registeroperanden, die nicht angegeben werden müssen.

Beispiele:

```
AAA          ;BENUTZT STETS REGISTER AL (IMPLIZIT)
MUL CL       ;AL IST ZIEL (IMPLIZIT), CL QUELLE
MOV AX,BX    ;AX IST ZIEL, BX IST QUELLE
ADD BL,ALPHA ;BL IST ZIEL
```

3.3. Unmittelbare Operanden

Der Quelloperand kann unmittelbar als Datenbyte oder -wort im Befehl angegeben werden. Auch hier sind keine Buszyklen erforderlich, der Operand hat einen festen Wert, der Bestandteil des Maschinenbefehls ist.
Unmittelbare Operanden können Zahlen im Binär-, Oktal-, Hexadezimal- oder Dezimalkode sowie ASCII-Zeichen sein. Sie können auch mit einer EQU-Anweisung als Konstanten vereinbart sein.

Beispiele:

```
MOV MASK,40H    ;LADE MASK MIT DEM WERT 40H
CMP AL,'E'      ;VERGLEICHE AL MIT 45H = ASCII-"E"
ADD BX,500      ;ADDIERE DIE ZAHL 500 ZU BX
CONST EQU 55
MOV CX,CONST    ;LADE CX MIT DEM WERT 55
```

3.4. Speicheroperanden

3.4.1. Datenformate

Speicheroperanden können Bytes, Worte und, bei einigen Befehlen, Doppelworte (Pointer) sein. Die Formate zeigt Bild 3.2.

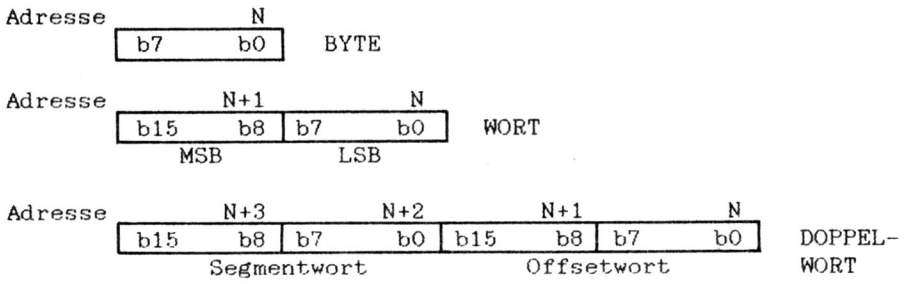


Bild 3.2. Formate der Speicheroperanden

3.4.2. Effektive Adresse

Die Bildung der Speicheradresse erfolgt aus der Sicht der Hardware durch Addition eines Offsets zu einer Basisadresse, welche durch Multiplikation eines Segmentregisters mit 16 gebildet wird. Der Offset ist für Befehlskodes im Register IP, für Stapeloperationen in SP enthalten. Für Speicheroperationen berechnet die Ausführungseinheit (EU) des K1810WM86 den Offset als "effektive Adresse" aus dem Befehlskode. Die effektive Adresse (EA) ist ein vorzeichenloser 16-Bit-Wert, der den Abstand in Byte von der Basisadresse des aktuellen Speichersegmentes angibt. Die effektive Adresse wird durch Addition des Inhalts eines Basisregisters (BX,BP), eines Indexregisters (DI,SI) und eines Displacements (konstanter Abstand) gebildet:

EA = BASISREGISTER + INDEXREGISTER + DISPLACEMENT

Dabei sind alle Kombinationen möglich, wodurch sich die vielfältigen Adressierungsverfahren nach Tafel 3.2 ergeben. Die Tafel enthält außerdem noch die Zeiten für die Berechnung der effektiven Adresse durch die EU.

Tafel 3.2. Adressierungsarten

Effektive Adresse	Verfahren	Takte
DISP	direkt	6
BX / DI / SI	indirekt	5
BX / BP + DISP	relativ	9
DI / SI + DISP	indiziert	9
BX / BP + DI / SI	basisindiziert	7 bis 8
BX / BP + DI / SI + DISP	basisindiziert	11 bis 13

Welche der Register zur Adreßbildung benutzt werden, wird im Befehlskode vorgegeben. Das Displacement ist ein fester Wert, der im Befehlskode enthalten ist. Es kann aus einem oder zwei Byte bestehen:

- 8-Bit-Displacement als vorzeichenbehaftete Zahl im Bereich
  -128 ... +127

- 16-Bit-Displacement als vorzeichenlose Zahl im Bereich
  0 ... 65535

Das Displacement wird bei der Generierung der Befehlskodes aus der Position des Operanden ermittelt. Außerdem kann es explizit im Befehl angegeben werden. Der K1810WM86 wendet immer ein 8-Bit-Displacement an, wenn es möglich ist. Bei der direkten Adressierung wird immer ein 16-Bit-Wert angegeben. (Hier sind Displacement und effektive Adresse identisch.)

3.4.3. Segmentüberschreibung

Bei der Adressierung von Speicherzellen gibt es normalerweise feste Zuordnungen zwischen Speichertyp (Kode, Stapel, Daten) und den Segmentregistern, die zur Bildung der physischen Adresse herangezogen werden. Der Programmierer muß sich nicht darum kümmern, da der K1810WM86 anhand des Befehlskodes automatisch die entsprechenden Segmentregister auswählt. Diese festen Zuordnungen zeigt Tafel 3.3. Es kann jedoch unter Umständen notwendig sein, einem Operanden ein anderes Segment zuzuordnen. Der Programmierer hat deshalb die Möglichkeit, eine Segmentüberschreibung anzugeben. Dazu wird das Segmentoverride-Präfix verwendet. Es hat die Form:

```
SR:OPERAND
```

Beispiele:

```
MOV AX,ES:BETA        ;BENUTZT ES ANSTELLE DS
CMP CS:[DI],DX        ;BENUTZT CS ANSTELLE DS
ADD DS:VAR[BP],4      ;BENUTZT DS ANSTELLE SS
```

Tafel 3.3. Anwendung der Segmentregister bei der Speicheradressierung

Speichertyp	Segment-register	alternative Segmentreg.	Offset
Befehlskode	CS	keine	IP
Daten (mit Ausnahme der folgenden)	DS	CS, ES, SS	EA
Stapeloperationen (PUSH, POP)	SS	keine	SP
Stringoperationen: Quelle	DS	CS, ES, SS	SI
Stringoperationen: Ziel	ES	keine	DI
Daten, die über BP adressiert werden	SS	CS, DS, SS	EA

3.4.4. Direkte Adressierung

Die effektive Adresse wird unmittelbar als Displacement im Befehlskode angegeben, es werden keine Register benutzt. Die direkte Adressierung benutzt normalerweise das Datensegmentregister DS zur Bildung der Segmentbasisadresse. Durch Segmentoverride-Präfixe können auch andere Segmentregister vereinbart werden.
Diese Adressierungsart wird benutzt, um einfache Variable (Skalare) zu adressieren (Bild 3.3).

Beispiele:

```
ADD BL,BETA        ;DER OFFSET VON BETA IST DIE EA
MOV OMEGA+8,AX     ;DER OFFSET VON OMEGA PLUS 8 IST
                   ;DIE EA
```

Der Assembler generiert bei dieser Adressierungsart immer ein 16-Bit-Displacement.

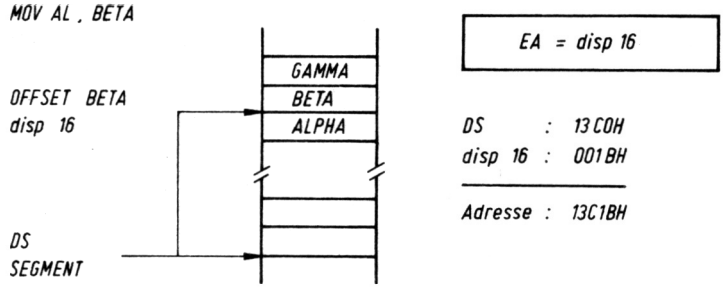


Bild 3.3. Direkte Adressierung

3.4.5. Indirekte Adressierung

Die effektive Adresse ist in einem der Register BX, DI oder SI enthalten; es wird kein Displacement verwendet. Darüber hinaus kann der Programmierer auch die indirekte Adressierung über das Register BP anwenden. Der Assembler generiert dann automatisch ein Displacementbyte mit dem Wert 00H, welches dem Maschinenkode angefügt wird (Bild 3.4).
Alle Variablen, die indirekt über DI, SI, BX adressiert werden, ordnet der K1810WM86 normalerweise dem Datensegment DS zu, während Adressen im Register BP dem Stapelsegment SS zugeordnet werden. Andere Segmentzuweisungen sind durch die Angabe eines Segmentoverride-Präfixes möglich.
Die indirekte Adressierung gestattet es, daß ein Befehl auf verschiedene Variablen zugreifen kann, wenn der Inhalt des entsprechenden Registers ständig aktualisiert wird. Dazu kann beispielsweise der Befehl LEA benutzt werden.

Beispiele:

```
ADD CX,[SI]      ;DIE EA IST IN SI ENTHALTEN
MOV [BP],SP      ;DIE EA IST IN BP ENTHALTEN
                 ;ES WIRD EIN DISPLACEMENT 0H GENERIERT
MOV AL,ES:[BX]   ;DIE EA BEZIEHT SICH AUF ES
```

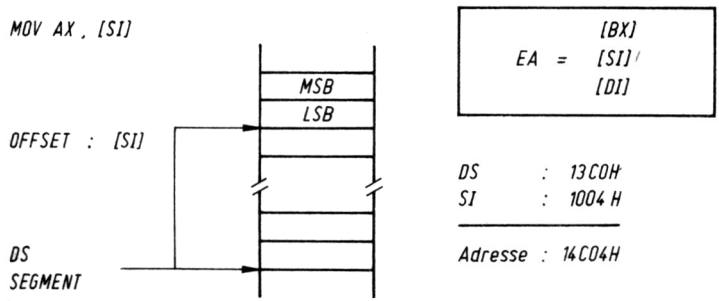


Bild 3.4. Indirekte Adressierung

3.4.6 Relative Adressierung

Bei diesem, auch als Basisadressierung bezeichneten Verfahren wird die effektive Adresse durch Addition eines Displacements mit dem Inhalt des BX- oder BP-Registers ermittelt. Dieses Register zeigt dabei auf den Anfang eines Speicherbereiches, während das Displacement den Abstand der Variablen relativ zu dieser Basis angibt. Bei Verwendung von BX werden dabei Daten im Datensegment (DS) adressiert, während BP dem Stapelsegment zugeordnet ist. Mit einem Segmentoverride-Präfix können diese Zuordnungen geändert werden.
Dieses Verfahren kann vorteilhaft zur Adressierung von Strukturen angewendet werden. Das sind Folgen von Speicherplätzen, denen eine feste Bedeutung zugewiesen wurde. Die Elemente einer Struktur können dabei unterschiedliches Format haben (Bytes, Worte, Doppelworte oder Arrays). Eine Struktur kann sich mehrfach im Speicher wiederholen oder aber verschieblich sein. Das Basisregister zeigt dann auf den jeweils aktuellen Anfang der Struktur, während das Displacement das konkrete Element adressiert (Bild 3.5).
Die Verwendung von BP gestattet es, auf Daten im Stapelsegment zuzugreifen, ohne den Stapelzeiger SP zu verändern.

```
Beispiel:      MOV AL,DELTA[BX]   ;DELTA IST ELEMENT DER DURCH BX
                                  ;ADRESSIERTEN STRUKTUR
```

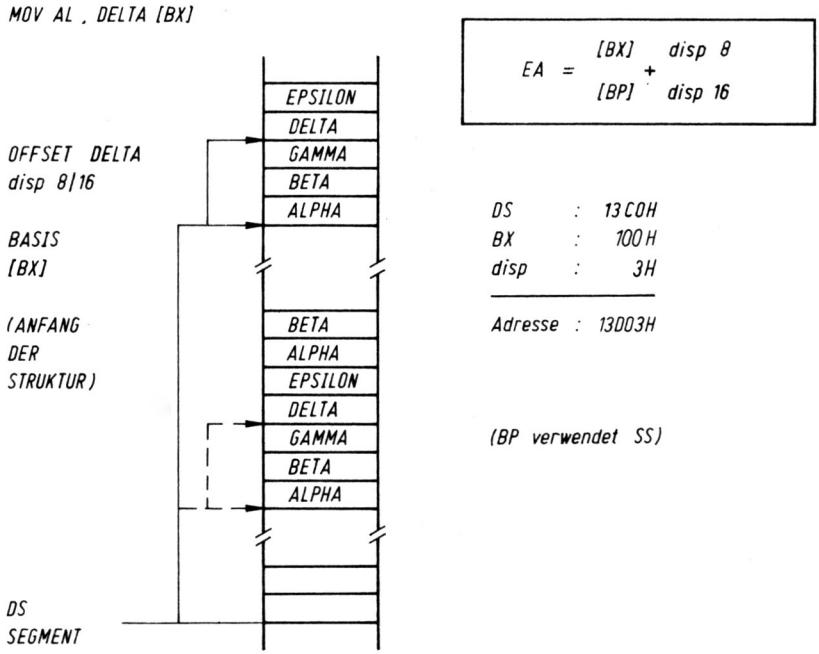


Bild 3.5. Relative Adressierung einer Struktur

3.4.7. Indizierte Adressierung

Die effektive Adresse wird durch Addition eines Displacements mit dem Inhalt eines Indexregisters DI oder SI gebildet. Auf Maschinenniveau ist dieses Verfahren also vergleichbar mit der relativen Adressierung. Ein Unterschied besteht jedoch in der Anwendung. Das Displacement zeigt auf den Anfang eines Speicherbereiches, während das Indexregister die Lage der Variablen relativ zu dieser Basis angibt. Normalerweise wird der Bereich im Datensegment (DS) lokalisiert, sofern kein Segmentoverride-Präfix angegeben ist.
Dieses Verfahren kann vorteilhaft zur Adressierung von Arrays (Feldern) angewendet werden. Das sind Folgen von Speicherplätzen, die eine Anordnung gleichartiger Elemente repräsentieren. Das Indexregister weist dabei auf das konkrete Element (es enthält dessen "Index"). Durch einfache arithmetische Manipulationen können alle Elemente mit einem Befehl erreicht werden (Bild 3.6).

```
Beispiel:        ADD AL,TABLE[SI]   ;TABLE IST EIN ARRAY AUS
                                    ;N ELEMENTEN
```

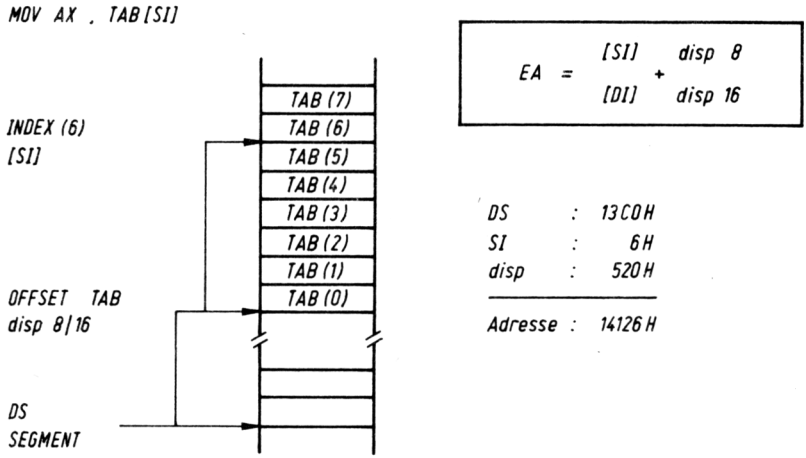


Bild 3.6. Indizierte Adressierung eines Arrays

3.4.8. Basisindizierte Adressierung

Dieses Verfahren ist eine Kombination aus relativer und indizierter Adressierung. Die effektive Adresse wird ermittelt durch Addition eines Displacements mit dem Inhalt eines Basis- und eines Indexregisters. Das Displacement kann auch entfallen; die Maschinenbefehle sind dann ein oder zwei Byte kürzer. Da bei diesem Verfahren zwei Adreßanteile gleichzeitig variiert werden können, ist es besonders zur Adressierung mehrdimensionaler Da-

tenstrukturen geeignet. Die Verwendung des BP-Registers ermöglicht dabei wiederum den Zugriff auf Daten im Stapelsegment (SS), während bei Verwendung von BX normalerweise auf das Datensegment (DS) zugegriffen wird.
Eine Anwendung ist die Adressierung von Feldelementen innerhalb einer Struktur oder einer Matrix (zweidimensionales Feld, Bild 3.7).

Beispiele:

```
MOV BX,PARAM[BP][SI] ;PARAM IST EIN FELD INNERHALB
                     ;EINER STAPELSTRUKTUR
CMP [BX][DI],10      ;2DIMENSIONALES ARRAY
```

Als alternative (assemblerabhängige) Schreibweisen werden auch folgende Formen angewendet:

```
MOV AL,ALPHA[BX+SI]
ADD [BP].BETA[DI],CX
```

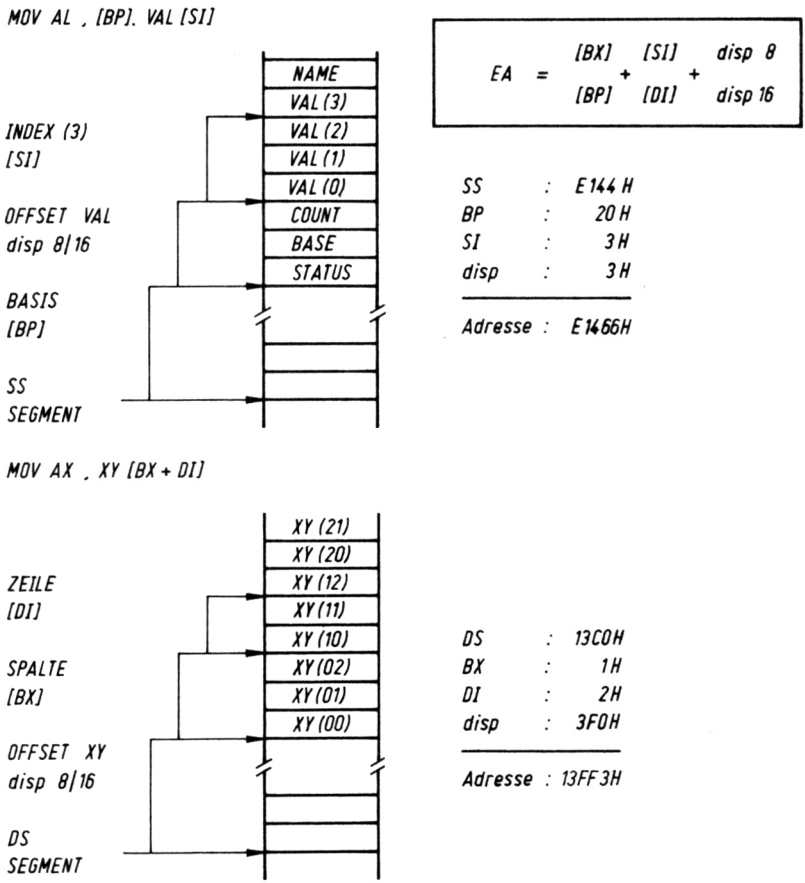


Bild 3.7. Basisindizierte Adressierung eines Arrays (oben) und einer Matrix (unten)

3.4.9. Stringadressierung

Bei Stringoperationen können - im Gegensatz zu anderen Befehlen des K1810WM86 - beide Operanden vom Typ Speicher sein. Deshalb wird hier ein besonderes Adressierungsverfahren angewendet. Die effektive Adresse des Quelloperanden ist stets in SI enthalten, während die des Zieloperanden in DI enthalten ist. Dieses Verfahren ist analog zur indirekten Adressierung; da aber kein alternatives Verfahren möglich ist, müssen keine Operanden angegeben werden. DI und SI werden nach jeder Übertragung automatisch aktualisiert.
Der Quelloperand wird DS zugeordnet; eine Segmentüberschreibung ist möglich. Der Zieloperand liegt stets in ES.

```
Beispiele:      MOVS DEST,SRC   ;DIE OPERANDEN DEST,SRC DIENEN NUR
                                ;DER TYPDEFINITION (BYTE ODER WORT)
                REP MOVSB       ;BYTETRANSFER MIT WIEDERHOLUNG
```

Verschiedene Stringoperationen benutzen auch den Akkumulator als Operand; der Speicher als zweiter Operand wird dennoch über DI/SI adressiert.

3.5. Adressierung von Ein-/Ausgabeports

Die Befehle IN und OUT übertragen Daten zwischen Akkumulator und einem Ein- oder Ausgabeport. Die Portadresse kann dabei als unmittelbarer 8-Bit-Wert angegeben werden oder im Register DX enthalten sein.
Ein-/Ausgabeports werden durch 16-Bit-Adressen ausgewählt. Bei der unmittelbaren Adressierung haben die Adreßbits 8 bis 15 den Wert Null, so daß diese Adressierungsart nur für 8-Bit-Adressierung (das höherwertige Byte wird ignoriert) oder die ersten 256 Portadressen bei 16-Bit-Adressierung angewendet werden kann.

```
Beispiele:      OUT C2H,AX      ;PORTADRESSE 00C2H
                IN AL,DX        ;PORTADRESSE VON 0 BIS FFFFH IN DX
```

3.6. Generierung der Maschinenbefehle

Die Maschinenbefehle des K1810WM86 sind durch eine byteorganisierte, variable Struktur gekennzeichnet. Sie können ein bis sechs Byte lang sein; ein Präfix bedingt noch ein zusätzliches Byte. Unabhängig von dieser Struktur liest der K1810WM86 den Kode wortweise ein. Damit werden sowohl die Zeit für das Befehlslesen als auch der Bedarf an Speicherplatz optimiert.
Einbytebefehle operieren grundsätzlich mit einem einzelnen Register oder den Flags (mit Ausnahme der Stringbefehle). Die Information zur Identifikation des Operanden ist unmittelbar im Kodebyte enthalten. Mehrbytebefehle operieren mit zwei Registern, mit Speicheroperanden oder unmittelbaren Werten. Ihre Struktur ent-

spricht typischerweise der im Bild 3.8 dargestellten.
Das Befehlsformat richtet sich nach bestimmten Eigenschaften des Befehls:

- Die ersten zwei Bytes definieren den Befehlstyp (Kode) und die Adressierungsart (Adreßmodus).
- Ist ein Displacement erforderlich, so folgt es dem Adreßmodusbyte. Ein 16-Bit-Displacement wird dabei in der Reihenfolge "niederwertiges (lo) Byte, höherwertiges (hi) Byte" angegeben.
- Ein unmittelbarer Operand folgt dem Displacement oder, falls der Adreßmodus keines erfordert, dem Adreßmodusbyte. Wortoperanden erscheinen in der Reihenfolge "niederwertiges (lo) Byte, höherwertiges (hi) Byte".

Das CODE-Byte besteht bei den meisten Befehlen aus 6 Bits, die den Befehlstyp bestimmen, und zwei Einzelbits. Diese können, falls ihnen kein fester Wert zugeordnet ist, folgende Bedeutung annehmen:

Bit 0 :	w = 0	Befehl behandelt Byte-Operanden
	w = 1	Befehl behandelt Wort-Operanden
Bit 1 :	d = 0	Das REG-Feld definiert den Quelloperanden
	d = 1	Das REG-Feld definiert den Zieloperanden.

Das Adreßmodusbyte spezifiziert die Adressierungsart. Es besteht aus drei Feldern:

- Das MD-Feld definiert den Typ der Adressierung und die Länge des Displacements (Tafel 3.4).
- Das REG-Feld spezifiziert den Registeroperanden, der je nach Wert des d-Bits Quelle oder Ziel, und, in Abhängigkeit vom w-Bit, ein 8- oder 16-Bit-Register sein kann. Manchmal wird auch ein Segmentregister angegeben (Tafel 3.5).
- Das R/M-Feld gibt in Verbindung mit MD entweder den zweiten Registeroperanden oder die Art der Berechnung der effektiven Adresse für einen Speicheroperanden an (Tafel 3.6).

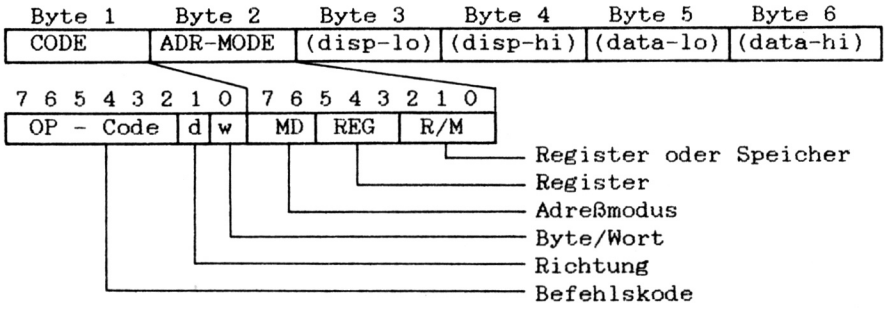


Bild 3.8. Struktur der Maschinenbefehle (Mehrbytebefehle)

Tafel 3.4. Das MD-Feld

MD	Bedeutung
00	Speicheradreßmodus ohne Displacement oder direkte Adressierung
01	Speicheradreßmodus, es folgt ein 8-Bit-Displacement
10	Speicheradreßmodus, es folgt ein 16-Bit-Displacement
11	Registermodus

Tafel 3.5. Das REG-Feld

REG	Bedeutung w = 0	w = 1	OSR	Bedeutung
000	AL	AX	000	ES
001	CL	CX	001	CS
010	DL	DX	010	SS
011	BL	BX	011	DS
100	AH	SP		
101	CH	BP		
110	DH	SI		
111	BH	DI		

Anmerkung: Bei Befehlen, die sich auf die Segmentregister beziehen, erhält das REG-Feld die Bedeutung OSR.

Tafel 3.6. Das R/M-Feld

R/M	Register MD = 11 w=0	w=1	Speicher MD = 00	MD = 01	MD = 10
000	AL	AX	BX+SI	BX+SI+DISP8	BX+SI+DISP16
001	CL	CX	BX+DI	BX+DI+DISP8	BX+DI+DISP16
010	DL	DX	BP+SI	BP+SI+DISP8	BP+SI+DISP16
011	BL	BX	BP+DI	BP+DI+DISP8	BP+DI+DISP16
100	AH	SP	SI	SI+DISP8	SI+DISP16
101	CH	BP	DI	DI+DISP8	DI+DISP16
110	DH	SI	DISP16	BP+DISP8	BP+DISP16
111	BH	DI	BX	BX+DISP8	BX+DISP16

Anmerkung: DISP16 ohne Registerangabe bedeutet direkte Adressierung.

Im Anhang C sind die Maschinenbefehle in numerischer Reihenfolge angegeben.

4. Befehlssatz des Mikroprozessors K1810WM86

In diesem Abschnitt wird der Befehlssatz des K1810WM86 vorgestellt. Die etwa 100 verschiedenen Assemblerbefehle sind nach Funktionsgruppen geordnet.

4.1. Übersicht

Die Assemblerbefehle des K1810WM86 werden in folgende Gruppen eingeteilt:

Datentransferbefehle:
Die Befehle dieser Gruppe übertragen Daten zwischen zwei Registern oder einem Register und dem Speicher, ohne die Operanden zu verändern. Eine besondere Befehlsklasse sind die Ein-/Ausgabebefehle, die dem Datentransfer im Ein-/Ausgabeadreßraum dienen.

Arithmetikbefehle:
In dieser Gruppe sind alle Befehle zur Ausführung von arithmetischen Operationen (Addition, Subtraktion, Multiplikation, Division) einschließlich Inkrementieren, Dekrementieren, Negieren und Vergleichen zusammengefaßt. Es können Binärzahlen mit und ohne Vorzeichen und Dezimalzahlen verarbeitet werden.

Bitmanipulations- und Logikbefehle:
Diese Befehle dienen der Ausführung logischer Operationen sowie dem Verschieben und Rotieren von Bits in Register- und Speicheroperanden.

Stringbefehle:
Die in dieser Gruppe zusammengefaßten Befehle dienen der Manipulation von Zeichenketten (Strings). Es können Übertragungs-, Vergleichs- und Suchoperationen mit Strings bis zu 64 KByte Länge ausgeführt werden.

Programmtransferbefehle:
Diese Gruppe enthält Befehle zur Realisierung von Programmverzweigungen (einschließlich Softwareschleifen) und dem Aufruf von Prozeduren. Neben einigen unbedingten Sprung- und Rufbefehlen verfügt der K1810WM86 über eine große Zahl bedingter Sprungbefehle. Eine besondere Klasse von Befehlen stellen die Software-Interrupts dar.

Prozessorsteuerbefehle:
Diese letzte Gruppe beinhaltet Befehle zur Flagbeeinflussung und zur Synchronisation der Programmabarbeitung mit externen Ereignissen.

Weiterhin existieren verschiedene Präfixe, die keine eigenständigen Befehle darstellen, sondern nur in Verbindung mit bestimmten Operationen verwendet werden können.

Die Befehle sind in den folgenden Beschreibungen mit den Mnemoniks bezeichnet, die von den meisten Assemblern angewendet werden. Für Operanden werden Pseudomnemoniks verwendet; das sind Symbole, die in den Assembleranweisungen durch entsprechende Ausdrücke ersetzt werden müssen:

src	:	Quelloperand (Register, Speicher oder Daten)
dest	:	Zieloperand (Register oder Speicher)
count	:	CL oder 1
procedure	:	Prozedur (Marke, Speicher- oder Registerpointer)
target	:	Zielbefehl (Marke, Speicher- oder Registerpointer)
typ	:	Typnummer (0 bis 255)
code	:	6-Bit-Kode für ESC-Befehle

Bei der Mehrzahl der Befehle werden einige Beispiele für mögliche Assembleranweisungen angegeben. Bei einfachen Befehlen, deren Anwendung klar ist, wird darauf verzichtet.

4.2. Datentransferbefehle

In dieser Befehlsgruppe werden alle Befehle zur Datenübertragung zwischen dem K1810WM86 und dem Speicher oder zwischen prozessorinternen Registern zusammengefaßt. Die Ein-/Ausgabebefehle werden als besondere Klasse mit eingeordnet.

4.2.1. Transferbefehle für allgemeine Anwendungen

Diese Klasse umfaßt vier Befehle, die Register- oder Speicherinhalte und unmittelbare Daten in Register oder Speicherplätze übertragen können. Die Daten werden dabei nicht verändert. Es werden keine Flags beeinflußt.

MOV dest,src

Übertrage ein Byte oder Wort

Dieser Befehl überträgt Byte- oder Wortoperanden zwischen Speicher und Register oder zwischen zwei Registern einschließlich Segmentregister. Das Laden von CS ist nicht erlaubt.

Beispiele:

```
MOV CX,AX                ;REGISTER NACH REGISTER
MOV AL,[BP]              ;SPEICHER NACH REGISTER
MOV WORD_VARIABLE,BX     ;REGISTER NACH SPEICHER
MOV ES,AX                ;REGISTER NACH SEGMENTREG
MOV SS,STACK_SEGMENT     ;SPEICHER NACH SEGMENTREG
MOV DI,ES                ;SEGMENTREG NACH REGISTER
MOV WORD PTR [BX],CS     ;SEGMENTREG NACH SPEICHER
MOV CX,10000             ;DATEN NACH REGISTER
MOV BYTE[DI],55H         ;DATEN NACH SPEICHER
MOV DS:[BP],1987         ;SEGMENTOVERRIDE
```

XCHG

Vertausche Quell- und Zieloperand

Der XCHG-Befehl vertauscht die Inhalte zweier Register oder eines Registers und eines Speicherplatzes miteinander. Die Segmentregister sind durch diesen Befehl nicht erreichbar.

```
Beispiele:      XCHG DI,SI            ;REGISTER MIT REGISTER
                XCHG SEMAPHORE,AL     ;REGISTER MIT SPEICHER
```

Der XCHG-Befehl bietet in Verbindung mit dem Lock-Präfix eine einfache Möglichkeit zum Testen und Setzen von Semaphoren. Das sind Speicherzellen, über die der Zugriff auf globale, von mehreren Prozessoren zur Kommunikation genutzte Speicherbereiche gesteuert wird (Bild 4.1).

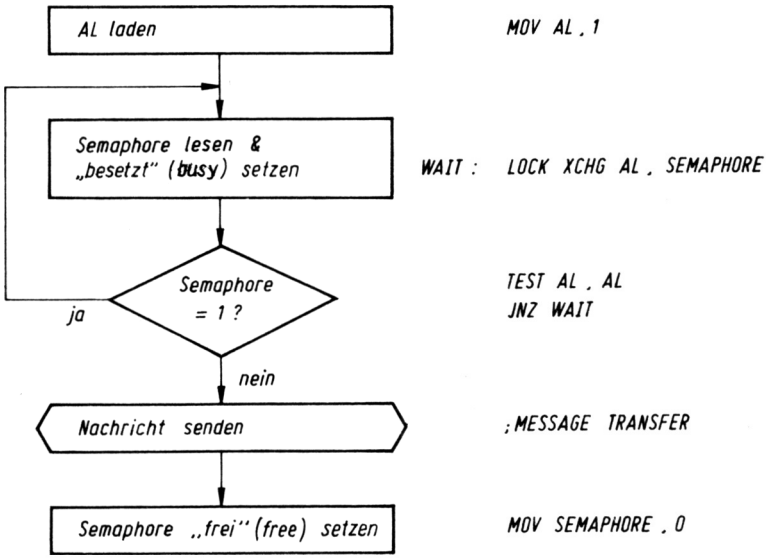


Bild 4.1. Anwendung von Semaphoren

XLAT / XLAT tab

Übersetze

Dieser besondere Befehl ersetzt den Inhalt des Akkumulators (AL) durch ein Byte aus einer vom Nutzer vorbereiteten 256 Byte großen Tabelle. Der Akkumulatorinhalt selbst dient dabei als Zeiger in diese Tabelle. Das BX-Register muß die Offsetadresse des ersten Bytes der Tabelle enthalten.

```
Beispiele:      XLAT
                XLAT CODE_TABLE
```

Der Befehl kann angewendet werden, wenn Zeichenketten aus einem Kode in einen anderen übersetzt werden sollen.
Der XLAT-Befehl wendet eine besondere Form der basisindizierten Adressierung an. Das BX-Register muß mit der Basisadresse der Tabelle geladen werden; AL wirkt als Index für jeweils eines der 256 Bytes. Wenn AL beispielsweise den Wert 0AH enthält, so wird dieser durch den Wert des zehnten Bytes der Tabelle ersetzt.
Es ist also keine Operandenangabe für die Berechnung der effektiven Adresse erforderlich. Der Operand "tab" kann aber als Variable vereinbart sein. Dann kann der Befehl LEA zum Laden von BX mit der richtigen Anfangsadresse benutzt werden.

PUSH src

Transportiere ein Wort in den Stapel

Dieser Befehl erniedrigt den Stapelzeiger (SP) um den Wert 2 und überträgt dann ein Wort auf die Spitze des Stapels (Top Of Stack), das heißt auf den durch SS:SP adressierten Speicherplatz. Damit ermöglicht dieser Befehl das Zwischenspeichern aller veränderlichen Daten (Registerinhalte, Parameter u.ä.) beim Aufruf von Unterprogrammen.

```
Beispiele:    PUSH SI              ;REGISTER
              PUSH ES              ;SEGMENTREGISTER
              PUSH PARAMETER[BX]   ;SPEICHERWORT
```

POP dest

Hole ein Wort aus dem Stapel

Dieser Befehl liest ein Wort aus dem durch SS:SP adressierten Speicherplatz (der aktuellen Stapelspitze) und überträgt es zum Zieloperanden. Anschließend wird der Stapelzeiger SP um den Wert 2 erhöht, so daß er auf die neue Stapelspitze zeigt. (Wir erinnern uns, daß der Stapel von oben beginnend in Richtung zu seiner Basis aufgebaut wird!) Mit Hilfe des POP-Befehls können am Ende eines Unterprogramms alle mit PUSH "geretteten" Daten wieder rekonstruiert werden.

```
Beispiele:    POP DI
              POP ALPHA[BX]
```

Dieser Befehl kann nicht auf CS angewendet werden, da in diesem Fall ein inkorrekter Programmsprung hervorgerufen wird. (Der Befehl PUSH segreg kann auf CS angewendet werden!)

4.2.2. Spezielle Befehle zum Laden von Adressen

In dieser Klasse sind drei Befehle zusammengefaßt, welche die Adressen von Variablen behandeln. Sie sind besonders günstig bei der Bearbeitung von Listen, Tabellen oder Zeichenketten anwendbar, das heißt, wenn Variable mittels Basis- und Indexregister adressiert werden.

LEA dest,src
Lade die effektive Adresse

Dieser Befehl lädt den Zieloperand, eines der 16-Bit-Register, mit der effektiven (Offset-) Adresse des Quelloperanden, der ein Speicheroperand (Variable) sein muß. Der Inhalt des Operanden selbst ist dabei ohne Interesse.

Beispiel: LEA DI,LABEL

In diesem Beispiel wird DI mit der Offsetadresse der Variablen LABEL geladen.

LDS / LES dest,src
Lade einen Zeiger in DS/ES

Die beiden Befehle LDS bzw. LES führen analoge Operationen aus. Sie laden eine Doppelwortvariable (32-Bit-Zeiger) aus dem Speicher in den Zieloperanden, der ein beliebiges 16-Bit-Register sein kann, und in eines der Segmentregister DS oder ES.
Die Zeigervariable (Pointer) wird zunächst wie eine normale Speichervariable behandelt, indem das niederwertigste Byte adressiert wird. Während aber bei normalen Speicheroperationen nur ein Byte oder Wort übertragen wird, werden bei Doppelwortvariablen zwei Worte (bzw. vier Bytes) übertragen. Das niederwertige Wort (Offsetwort) wird in das Zielregister übertragen, das höherwertige (Segmentwort) in das Segmentregister.

Beispiele:
```
LDS SI,SOURCE_STRING
LES DI,DEST_STRING
```

Diese Befehle sind geeignet, Stringoperationen vorzubereiten. Die Zeigervariablen SOURCE_STRING und DEST_STRING (Doppelworte!) enthalten die aktuellen Adressen (Segment und Offset), und mit den genannten Befehlen können alle Register korrekt initialisiert werden.

4.2.3. Befehle zur Flagübertragung

Diese Befehle dienen der Übertragung des Programmstatuswortes (PSW) oder bestimmter Flags. Ziel oder Quelle kann dabei nur der Stapel oder das Register AH sein. Die Flags werden dabei in folgendem Format transportiert:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Flag:	U	U	U	U	O	D	I	T	S	Z	U	A	U	P	U	C

U	undefiniert	
O	Überlaufflag	(Overflow)
D	Richtungsflag	(Direction)
I	Unterbrechungsflag	(Interrupt)
T	Testflag	(Trap)
S	Vorzeichenflag	(Sign)
Z	Nullflag	(Zero)
A	Hilfsübertragsflag	(Auxiliary Carry)
P	Paritätsflag	(Parity)
C	Übertragsflag	(Carry)

Bei Übertragungen nach oder aus AH wird nur das niederwertige Byte, also S, Z, A, P und C transportiert. Bei Stapelübertragungen wird das gesamte Wort transportiert.

LAHF

Lade das Register AH mit den Flags

Dieser Befehl kopiert die Flags S, Z, A, P und C in das Register AH, wobei die Flags selbst unverändert bleiben.

SAHF

Übertrage das Register AH in die Flags

Dieser Befehl kopiert die Bits 0, 2, 4, 6 und 7 in die entsprechenden Bits des Programmstatuswortes. Damit geht der alte Zustand der Flags S, Z, A, P und C verloren.

PUSHF

Transportiere die Flags in den Stapel

Dieser Befehl erniedrigt den Stapelzeiger (SP) um den Wert 2 und überträgt dann das Programmstatuswort auf die aktuelle Stapelspitze, das heißt in die durch SS:SP adressierte Speicherzelle Während PUSH dem Retten allgemeiner Daten und Registerinhalte dient, kann mit PUSHF der Zustand der Flags beim Sprung in ein Unterprogramm gerettet werden.
Beim Aufruf einer Interruptprozedur werden die Flags automatisch in den Stapel übertragen!

POPF

Hole die Flags aus dem Speicher

Dieser Befehl überträgt das an der aktuellen Stapelspitze abgespeicherte Wort in das Programmstatuswort. Anschließend wird der Stapelzeiger (SP) um den Wert 2 erhöht (siehe auch bei POP).
Dieser Befehl wirkt analog dem allgemeinen POP-Befehl und dient dem Rekonstruieren des alten Zustandes der Flags bei der Rückkehr aus einem Unterprogramm.
Bei der Rückkehr aus einer Interruptprozedur werden die Flags automatisch aus dem Stapel geholt!

Beispiel:

```
;DIE FOLGENDE PROZEDUR SETZT DAS T-FLAG
PUSHF
POP AX          ;FLAGS NACH AX KOPIEREN
OR AX,100H      ;BIT 8 SETZEN
PUSH AX
POPF
```

4.2.4. Ein-/Ausgabebefehle

Der K1810WM86 verfügt über zwei Befehle zur Übertragung von Daten zu oder von Ein-/Ausgabeports. Die beiden Befehle IN und OUT übertragen Daten nur zwischen Akkumulator und Port.
Die Portadresse kann für die Ports 0 bis 255 (Adressen 0 bis 0FFH) direkt im Befehl angegeben werden, ansonsten muß sie im Register DX enthalten sein.

OUT dest,src

Gib ein Byte oder Wort an ein Ausgabeport aus

Der Befehl OUT überträgt Byte- oder Wortdaten vom Akkumulator zu einem Ausgabeport.

Beispiele:

```
(1) OUT 0C4H,AL      ;DIREKTE PORTADRESSE

(2) MOV DX,PORT_ADR
    OUT DX,AX
```

IN dest,src

Lies ein Byte oder Wort von einem Eingabeport

Der Befehl IN überträgt Byte- oder Wortdaten von einem Eingabeport zum Akkumulator.

Beispiele:

```
(1) IN AX,20         ;DIREKTE PORTADRESSE

(2) MOV DX,PORT_ADR
    IN AL,DX
```

4.3. Arithmetikbefehle

Der K1810WM86 bietet einen Satz von Befehlen für alle vier mathematischen Grundoperationen Addition, Subtraktion, Multiplikation und Division an. Es können 8- und 16-Bit-Operationen mit vorzeichenlosen und vorzeichenbehafteten Binärzahlen durchgeführt werden. Darüber hinaus können auch Dezimalzahlen verarbeitet werden.

4.3.1. Zahlenformate

Bei den Arithmetikbefehlen können sechs verschiedene Zahlenformate angewendet werden:

Vorzeichenlose 8-Bit-Binärzahlen
Zahlenbereich: 0 bis 255
Format:

b7	b6	b5	b4	b3	b2	b1	b0

Vorzeichenlose 16-Bit-Binärzahlen
Zahlenbereich: 0 bis 65535
Format:

b15	b14	b13	b12	b11	b10	b9	b8
b7	b6	b5	b4	b3	b2	b1	b0

Vorzeichenlose 8- oder 16-Bit-Binärzahlen können ohne Einschränkung bei allen vier Rechenoperationen angewendet werden.

Vorzeichenbehaftete 8-Bit-Binärzahlen (Byte Integer)
Zahlenbereich: -128 bis +127
Format:

s	b6	b5	b4	b3	b2	b1	b0

s = 0 : positiv
s = 1 : negativ , Darstellung im Zweierkomplement

Vorzeichenbehaftete 16-Bit-Binärzahlen (Word Integer)
Zahlenbereich: -32768 bis +32767
Format:

s	b14	b13	b12	b11	b10	b9	b8
b7	b6	b5	b4	b3	b2	b1	b0

s = 0 : positiv
s = 1 : negativ , Darstellung im Zweierkomplement

Vorzeichenbehaftete Binärzahlen (Integers) können ebenfalls bei allen vier Rechenoperationen angewendet werden. Durch die Darstellung negativer Zahlen im Zweierkomplement sind für Addition und Subtraktion keine speziellen Befehle erforderlich, das heißt, Zahlen mit und ohne Vorzeichen werden mit den gleichen Befehlen behandelt. Für Multiplikation und Division enthält der Befehlssatz spezielle Befehle.

Gepackte Dezimalzahl (gepackt BCD)
Zahlenbereich: 0 bis 99
Format:

y3	y2	y1	y0	x3	x2	x1	x0

y : höherwertiges Digit (Zehner) , binär kodiert
x : niederwertiges Digit (Einer) , binär kodiert

Gepackte Dezimalzahlen werden als vorzeichenlose 8-Bit-Werte behandelt. Jedes Halbbyte (Nibble) repräsentiert ein BCD-Digit. Die Anwendung ist nur bei Addition und Subtraktion möglich, wobei die Operation stets in zwei Etappen durchgeführt wird:

- Eine binäre Addition oder Subtraktion der beiden BCD-Zahlen liefert zunächst ein Zwischenergebnis.
- Ein anschließender Justagebefehl (DAA, DAS) wandelt dieses Zwischenergebnis in eine gepackte Dezimalzahl um.

Ungepackte Dezimalzahl (ungepackt BCD)
Zahlenbereich: 0 bis 9
Format:

0	0	0	0	x3	x2	x1	x0

x : binär kodiertes Digit

Ungepackte Dezimalzahlen können bei allen vier Rechenoperationen angewendet werden. Sie werden als vorzeichenlose 8-Bit-Werte behandelt, wobei das niederwertige Halbbyte (Nibble) das BCD-Digit enthält (Hexadezimalwert 0 bis 9). Das höherwertige Halbbyte wird bei Addition und Subtraktion ignoriert, bei Multiplikation und Division muß es auf Null gesetzt sein.
Operationen mit ungepackten Dezimalzahlen erfolgen in zwei Schritten:

- Bei Addition, Subtraktion und Multiplikation liefert zunächst eine binäre Operation ein Zwischenergebnis.
- Ein anschließender Justagebefehl (AAA, AAS, AAM) wandelt dann das Zwischenergebnis in ungepacktes BCD-Format um.
- Bei der Division muß zunächst der Dividend durch den AAD-Befehl korrigiert werden, die anschließende binäre Division mit einer BCD-Zahl liefert ein korrektes Ergebnis im ungepackten BCD-Format.

Das ungepackte BCD-Format gestattet auch das Rechnen mit ASCII-Ziffern. (Der ASCII-Kode entspricht dem als RGW-Standard festgelegten KOI-7-Kode.) Es ist jedoch zu beachten, daß bei dieser Zeichendarstellung in den Ziffern 0 bis 9 das höherwertige Halbbyte auf 3H gesetzt ist. Um ASCII-Ziffern als Ergebnis zu erhalten, muß vor MUL- und DIV-Befehlen das höherwertige Halbbyte der Operanden auf Null und im Ergebnis nach allen Operationen wieder auf 3H gesetzt werden.

4.3.2. Flagbeeinflussung

Die Arithmetikbefehle beeinflussen je nach Ergebnis folgende Flags:

O	Überlaufflag	(Overflow)
S	Vorzeichenflag	(Sign)
Z	Nullflag	(Zero)
A	Hilfsübertragsflag	(Auxiliary Carry)
P	Paritätsflag	(Parity)
C	Übertragsflag	(Carry)

Tafel 4.1 zeigt, welche Flags durch die einzelnen Befehle dieser Gruppe beeinflußt werden.

Tafel 4.1. Beeinflussung der Flags bei Arithmetikoperationen

Befehl		O	S	Z	A	P	C
Addition:	ADD, ADC	X	X	X	X	X	X
	INC	X	X	X	X	X	-
	AAA	U	U	U	X	U	X
	DAA	U	X	X	X	X	X
Subtraktion:	SUB, SBB	X	X	X	X	X	X
	NEG, CMP	X	X	X	X	X	X
	DEC	X	X	X	X	X	-
	AAS	X	U	U	X	X	X
	DAS	U	X	X	X	X	X
Multiplikation:	MUL, IMUL	X	U	U	U	U	X
	AAM	U	X	X	U	X	U
Division:	DIV, IDIV	U	U	U	U	U	U
	AAD	U	X	X	U	X	U
	CBW, CWD	-	-	-	-	-	-

X : Flag wird aktualisiert
- : Flag bleibt unbeeinflußt
U : Flag nimmt undefinierten Zustand ein

Für die Beeinflussung der Flags durch Arithmetikoperationen gelten folgende Regeln:

1. Das Übertragsflag C wird auf 1 gesetzt, wenn bei einer Addition ein Übertrag aus dem höchstwertigen Bit (Carry) oder bei einer Subtraktion ein Borgen in das höchstwertige Bit (Borrow) entsteht.
   Es zeigt einen Überlauf vorzeichenloser Zahlen an. Bei der Multiplikation wird C gemeinsam mit O auf 1 gesetzt, wenn das Ergebnis einer Bytemultiplikation ein Wort oder bei einer Wortmultiplikation ein Doppelwort ist, ansonsten sind beide Flags 0.

2. Das Hilfsübertragsflag A wird auf 1 gesetzt, wenn ein Übertrag aus dem oder in das niederwertigste Halbbyte entsteht.
   Es wird in den Befehlen AAA, DAA, AAS und DAS getestet, um korrekte Ergebnisse bei Operationen mit Dezimalzahlen zu erhalten.

3. Das Nullflag Z wird auf 1 gesetzt, wenn das Ergebnis einer Operation Null ist.

4. Das Vorzeichenflag S wird auf 1 gesetzt, wenn im Ergebnis einer Operation das höchstwertige Bit (entweder Bit 7 oder Bit 15) den Wert 1 hat.
   Bei vorzeichenbehafteten Zahlen enthält es also immer das Vorzeichen des Ergebnisses (S=0: positiv, S=1: negativ).

5. Das Paritätsflag P wird auf 1 gesetzt, wenn das niederwertige Byte des Ergebnisses eine gerade Anzahl Einsen enthält (gerade Parität). Bei einer ungeraden Anzahl Einsen (ungerade Parität) wird P auf 0 gesetzt.
   Dieses Flag kann zum Beispiel zum Testen von ASCII-Zeichen angewendet werden.

6. Das Überlaufflag O wird auf 1 gesetzt, wenn bei einer Addition ein Übertrag vom nächstniederen in das höchstwertige Bit entsteht (d.h von Bit 6 nach Bit 7 bei Bytes oder von Bit 14 nach Bit 15 bei Worten), ohne daß ein Übertrag aus dem höchstwertigen Bit heraus auftritt, oder wenn bei einer Subtraktion ein Borgen vom höchstwertigen in das nächstniedere Bit entsteht, ohne daß ein Borgen in das höchstwertige Bit auftritt.
   Dieses Flag zeigt somit einen Überlauf bei vorzeichenbehafteten Zahlen an, das heißt, das Ergebnis überschreitet den zulässigen Wertebereich.
   Bei der Multiplikation wird O gemeinsam mit C auf 1 gesetzt, wenn das Ergebnis einer Bytemultiplikation ein Wort oder bei einer Wortmultiplikation ein Doppelwort ist; ansonsten sind beide Flags 0.

Alle Flags außer A können mit bedingten Sprungbefehlen getestet werden (siehe Abschn. 4.6). Das Überlaufflag O kann außerdem durch den Befehl INTO (Interrupt On Overflow) behandelt werden.

4.3.3. Addition

Die Befehle ADD und ADC ermöglichen die Additon von Binärzahlen, wobei ADC das Übertragsflag C mit berücksichtigt. Durch die Darstellung negativer Zahlen im Zweierkomplement können diese Befehle sowohl für vorzeichenlose als auch für vorzeichenbehaftete Zahlen angewendet werden. Sollen BCD-Zahlen addiert werden, so muß eine Ergebniskorrektur mit den Befehlen AAA oder DAA durchgeführt werden. Ein spezieller Befehl dient dem Inkrementieren (Addition von Eins).

ADD dest,src

Addiere

Dieser Befehl führt eine binäre Addition von Quell- und Zieloperand aus. Der Zieloperand wird durch das Ergebnis der Addition ersetzt. Es können Registerwerte miteinander, Register- mit Speicherwerten oder Konstanten mit Register- oder Speicherwerten addiert werden.

Beispiele:

```
ADD AL,BL                ;REGISTER + REGISTER
ADD DX,DELTA             ;REGISTER + SPEICHER
ADD BETA[BX],CL          ;SPEICHER + REGISTER
ADD AL,10H               ;REGISTER + DATEN
ADD WORD PTR [BX],-100   ;SPEICHER + DATEN
```

ADC dest,src

Addiere mit Übertrag

Dieser Befehl führt eine binäre Addition von Quell- und Zieloperand aus. Falls das Übertragsflag C gesetzt ist, wird dazu Eins addiert. Der Zieloperand wird durch das Ergebnis der Addition ersetzt. Dieser Befehl ist geeignet, Programme zur Addition von 32- oder 64-Bit-Zahlen zu schreiben, da er einen Übertrag aus der jeweils vorangegangenen Operation berücksichtigt.
Es können Registerwerte miteinander, Register- mit Speicherwerten oder Konstanten mit Register- oder Speicherwerten addiert werden.

INC dest

Inkrementiere um 1

Der Befehl INC erhöht den Operanden um den Wert 1. Der Operand wird dabei als vorzeichenlose Zahl behandelt. Es können Register- und Speicherinhalte inkrementiert werden.

AAA

Korrigiere ein Byte in eine ungepackte BCD-Zahl nach der Addition

Dieser Befehl dient der Umwandlung von 8-Bit-Binärzahlen in ungepacktes BCD-Format nach einer vorangegangenen Addition. Dadurch kann die Addition ungepackter Dezimalzahlen oder numerischer ASCII-Zeichen realisiert werden. Dabei wird zunächst eine binäre Byteaddition der beiden Dezimalzahlen mit dem Register AL als Zieloperand ausgeführt. Der anschließende Befehl AAA konvertiert dann das Zwischenergebnis in eine ungepackte Dezimalzahl.

Beispiel:

```
MOV AL,'3'
ADD AL,ASCII_NUMERAL
AAA
```

Nach dem AAA-Befehl enthält das AL-Register eine ungepackte Dezimalzahl zwischen 0 und 9. Wurde vor der Addition AH auf 0H gesetzt, so enthält es den Wert 1, falls die Summe der beiden Operanden größer als 9 ist.

DAA

Korrigiere ein Byte in eine gepackte BCD-Zahl nach der Addition

Dieser Befehl dient der Umwandlung von 8-Bit-Binärzahlen in gepacktes BCD-Format nach einer vorangegangenen Addition. Dadurch kann die Addition zweistelliger gepackter Dezimalzahlen realisiert werden. Dabei wird zunächst eine binäre Byteaddition der beiden Dezimalzahlen mit dem Register AL als Zieloperand ausgeführt. Der anschließende Befehl DAA konvertiert dann das Zwischenergebnis in eine gepackte Dezimalzahl.

Beispiel:

```
MOV AL,3
ADC AL,DECIMAL_VALUE
DAA
```

Nach dieser Operation enthält das AL-Register eine gepackte Dezimalzahl zwischen 0 und 99. Ist das Ergebnis größer, so wird das C-Flag gesetzt.

4.3.4. Subtraktion

Die Befehle SUB und SBB ermöglichen die Subtraktion von Binärzahlen mit und ohne Vorzeichen.
Für Operationen mit Dezimalzahlen stehen die Befehle AAS und DAS zur Ergebniskorrektur zur Verfügung. Zwei spezielle Befehle dienen dem Dekrementieren und Negieren von Operanden, außerdem ist ein arithmetischer Vergleichsbefehl implementiert.

SUB dest,src

Subtrahiere

Dieser Befehl führt eine binäre Subtraktion des Quell- vom Zieloperanden aus. Der Zieloperand wird durch das Ergebnis der Subtraktion ersetzt. Es können Registerwerte voneinander, Register- von Speicherwerten und umgekehrt oder Konstanten von Register- oder Speicherwerten subtrahiert werden.

Beispiele:

```
SUB AL,BL                ;REGISTER - REGISTER
SUB DX,ALPHA             ;REGISTER - SPEICHER
SUB BETA[BX],CL          ;SPEICHER - REGISTER
SUB CX,5                 ;REGISTER - DATEN
SUB BYTE PTR [DI],100    ;SPEICHER - DATEN
```

SBB dest,src

Subtrahiere mit Borgen

Dieser Befehl führt eine binäre Subtraktion von Quell- und Zieloperand aus. Falls das Übertagsflag C gesetzt ist, wird vom Ergebnis Eins subtrahiert. Der Zieloperand wird durch das Ergebnis der Subtraktion ersetzt. Dieser Befehl ist geeignet, Programme zur Subtraktion von 32- oder 64-Bit-Zahlen zu schreiben, da er einen Übertrag aus der jeweils vorangegangenen Operation berücksichtigt.
Es können Registerwerte voneinander, Register- von Speicherwerten und umgekehrt oder Konstanten von Register- oder Speicherwerten subtrahiert werden.

DEC dest

Dekrementiere um 1

Der Befehl DEC erniedrigt den Operanden um den Wert Eins. Der Operand wird dabei als vorzeichenlose Zahl behandelt. Es können Register- und Speicherinhalte dekrementiert werden.

NEG dest

Negiere

Der Befehl NEG bildet das Zweierkomplement des Zieloperanden. Das bedeutet, daß das Vorzeichen einer Zahl invertiert wird. Die Negation von Null setzt das Übertragsflag C auf 0, ansonsten wird C auf 1 gesetzt. Die Bytenegation von -128 bzw. die Wortnegation von -32768 ist nicht möglich; dabei bleibt der Operand unverändert, und das Überlaufflag O wird auf 1 gesetzt. Es können Register- und Speicherinhalte negiert werden.

CMP dest,src

Vergleiche

Dieser Befehl führt eine binäre Subtraktion des Quell- vom Zieloperanden aus. Das Ergebnis wird jedoch nicht abgespeichert. Die Operanden bleiben also unverändert, es werden nur die Flags entsprechend dem Ergebnis der Subtraktion aktualisiert. Durch nachfolgende bedingte Sprungbefehle können auf diese Weise zwei Zahlen auf arithmetische Bedingungen, wie "gleich", "größer" oder "kleiner", getestet werden.
Es können Registerwerte miteinander, Register- mit Speicherwerten oder Konstanten mit Register- oder Speicherwerten verglichen werden.

Beispiel:

```
CMP CX,ES:LIMIT
JLE BEGIN
```

AAS

Korrigiere ein Byte in eine ungepackte BCD-Zahl nach der Subtraktion

Dieser Befehl dient der Umwandlung von 8-Bit-Binärzahlen in ungepacktes BCD-Format nach einer vorangegangenen Subtraktion. Dadurch kann die Subtraktion ungepackter Dezimalzahlen oder numerischer ASCII-Zeichen realisiert werden. Dabei wird zunächst mit einem Befehl SUB oder SBB eine binäre Bytesubtraktion der beiden Dezimalzahlen mit dem Register AL als Zieloperand ausgeführt. Der anschließende Befehl AAS konvertiert dann das Zwischenergebnis in eine ungepackte Dezimalzahl.

Beispiel:
```
SUB AL,ASCII_NUMERAL
AAS
```

Nach dem AAS-Befehl enthält das AL-Register eine ungepackte Dezimalzahl zwischen 0 und 9.

DAS

Korrigiere ein Byte in eine gepackte BCD-Zahl nach der Subtraktion

Dieser Befehl dient der Umwandlung von 8-Bit-Binärzahlen in gepacktes BCD-Format nach einer vorangegangenen Subtraktion. Dadurch kann die Subtraktion zweistelliger gepackter Dezimalzahlen realisiert werden. Dabei wird zunächst mit einem Befehl SUB oder SBB eine binäre Bytesubtraktion der beiden Dezimalzahlen mit dem Register AL als Zieloperand ausgeführt. Der anschließende Befehl DAS konvertiert dann das Zwischenergebnis in eine gepackte Dezimalzahl.

Beispiel:
```
SBB AL,DECIMAL_VALUE
DAS
```

Nach dem DAS-Befehl enthält das AL-Register eine gepackte Dezimalzahl.

4.3.5. Multiplikation

Der K1810WM86 kann vorzeichenlose und vorzeichenbehaftete 8- und 16-Bit-Binärzahlen multiplizieren. Dafür werden zwei verschiedene Befehle angewendet. Es können Speicher- und Registerinhalte mit dem Akkumulatorinhalt multipliziert werden. Dabei entsteht ein Ergebnis von doppelter Länge. Der niederwertige Teil des Ergebnisses steht bei Bytemultiplikationen in AL, bei Wortmultiplikationen in AX. Der höherwertige Teil steht im Akkumulatorerweiterungsregister AH bei Byte- oder DX bei Wortoperationen.

MUL src

Multipliziere vorzeichenlose Zahlen

Dieser Befehl dient der Multiplikation vorzeichenloser Byte- und Wortoperanden. Ist der Quelloperand ein Byte, so wird er mit dem Inhalt von AL multipliziert und das 16-Bit-Ergebnis in AX abgespeichert. Ist der Quelloperand ein Wort, so wird er mit dem Inhalt von AX multipliziert und das 32-Bit-Ergebnis in AX (niederwertiges Wort) und DX (höherwertiges Wort) abgespeichert. Falls der jeweils höherwertige Teil signifikant, das heißt von Null verschieden ist, werden das Übertrags- und das Überlaufflag (C und O) auf 1 gesetzt.

Beispiele:
```
(1) MOV BL,FAKTOR_1
    MOV AL,FAKTOR_2
    MUL BL              ;PRODUKT IN AH,AL

(2) MOV AX,1000
    MUL WORD_VALUE      ;PRODUKT IN DX,AX
```

IMUL src

Multipliziere vorzeichenbehaftete Zahlen

Dieser Befehl dient der Multiplikation vorzeichenbehafteter Byte- und Wortoperanden. Ist der Quelloperand ein Byte, so wird er mit dem Inhalt von AL multipliziert und das 16-Bit-Ergebnis in AX abgespeichert. Ist der Quelloperand ein Wort, so wird er mit dem Inhalt von AX multipliziert und das 32-Bit-Ergebnis in AX (niederwertiges Wort) und DX (höherwertiges Wort) abgespeichert. Falls der jeweils höherwertige Teil signifikant ist, werden das Übertrags- und das Überlaufflag (C und O) auf 1 gesetzt. Sind beide Flags auf 0 gesetzt, so enthalten die Erweiterungsregister die Vorzeichenverlängerung der niederwertigen Hälfte des Ergebnisses, das heißt, alle Bits von AH bzw. DX sind 0 bei positivem und 1 bei negativem Ergebnis.

Beispiele:
```
(1) MOV AL,-10
    IMUL INTEGER        ;PRODUKT IN AH,AL

(2) IMUL BX             ;PRODUKT IN DX,AX
```

AAM

Korrigiere das Ergebnis in eine ungepackte BCD-Zahl nach der Multiplikation

Mit diesem Befehl kann das Ergebnis einer Multiplikation zweier ungepackter BCD-Zahlen (Bytes) in ungepacktes BCD-Format korrigiert werden. Um ein korrektes Ergebnis zu erhalten, müssen die höherwertigen Halbbytes der beiden Operanden vor der Multiplikation auf 0H gesetzt sein. Nach Ausführung einer binären Bytemultiplikation mit dem MUL-Befehl erhält man zunächst ein binäres

Zwischenergebnis. Dieses wird anschließend mit dem AAM-Befehl in eine ungepackte Dezimalzahl konvertiert. Das niederwertige Digit (Einer) steht in AL, das höherwertige (Zehner) in AH.

Beispiel:
```
MOV BL,7
MOV AL,5
MUL BL          ;DAS ERGEBNIS IN AL IST 23H
AAM             ;DAS ERGEBNIS IN AX IST 0305H
                ;ES ENTSPRICHT DER DEZIMALZAHL 35
                ;IN UNGEPACKTER BCD-DARSTELLUNG
```

4.3.6. Division

Der K1810WM86 bietet die Möglichkeit, vorzeichenlose und vorzeichenbehaftete Binärzahlen zu dividieren. Ein Korrekturbefehl erlaubt darüber hinaus die Division ungepackter Dezimalzahlen. Als Divisor können 8- oder 16-Bit-Register- oder Speicherinhalte benutzt werden. Der Dividend von doppelter Länge muß im Akkumulator und einem Erweiterungsregister stehen (bei Bytedivisionen sind das AL und AH, bei Wortdivisionen AX und DX). Zwei spezielle Befehle erlauben die Konvertierung von Bytes in Worte oder von Worten in Doppelworte, um korrekte Dividenden zu bilden.

DIV src

Dividiere vorzeichenlose Zahlen

Dieser Befehl dividiert den erweiterten Akkumulatorinhalt durch eine vorzeichenlose 8- oder 16-Bit-Zahl. Ist der Quelloperand ein Byte, so wird als Dividend das in AH;AL enthaltene Wort benutzt. Das Ergebnis wird in AL abgespeichert (8-Bit-Quotient), während AH den bei der Division entstandenen Rest enthält. Ist der Quelloperand ein Wort, so wird als Dividend das in DX;AX enthaltene Doppelwort benutzt. Der 16-Bit-Quotient wird in AX, der Rest in DX abgespeichert. Der Quotient wird dabei stets zu einer ganzen Zahl abgerundet.
Ist der Quotient bei einer Bytedivision größer als 255 oder bei einer Wortdivision größer als 65535, so daß er nicht mehr im Akkumulator dargestellt werden kann, so wird ein Interrupt vom Typ 0 ausgelöst (Divisionsfehler, siehe Abschn. 2.5.3).
Ein solcher Fall tritt bei einer Division durch Null auf.

Beispiele:
```
(1) MOV DX,HI_WORD
    MOV AX,LO_WORD
    MOV BX,DIVISOR_WORD
    DIV BX              ;QUOTIENT IN AX, REST IN DX

(2) MOV AX,DIVIDEND
    DIV BYTE PTR [SI] ;QUOTIENT IN AL, REST IN AH
```

IDIV src

Dividiere vorzeichenbehaftete Zahlen

Dieser Befehl dividiert den erweiterten Akkumulatorinhalt durch eine vorzeichenbehaftete 8- oder 16-Bit-Zahl. Ist der Quelloperand ein Byte, so wird als Dividend das in AH;AL enthaltene Wort benutzt. Das Ergebnis wird in AL abgespeichert (8-Bit-Quotient), während AH den bei der Division entstandenen Rest enthält. Ist der Quelloperand ein Wort, so wird als Dividend das in DX;AX enthaltene Doppelwort benutzt. Der 16-Bit-Quotient wird in AX, der Rest in DX abgespeichert. Quotient und Rest sind ebenfalls vorzeichenbehaftete Zahlen, beide haben das gleiche Vorzeichen. Positive Quotienten werden stets abgerundet, negative hingegen aufgerundet, das heißt, die Rundung erfolgt stets in Richtung Null.
Ist der Quotient bei einer Bytedivision größer als +127 oder kleiner als -128 oder bei einer Wortdivision größer als +32767 oder kleiner als -32768, so daß er nicht mehr im Akkumulator dargestellt werden kann, so wird ein Interrupt vom Typ 0 ausgelöst (Divisionsfehler, siehe Abschn. 2.5.3).
Ein solcher Fall tritt bei einer Division durch Null auf.

Beispiele:

```
(1) MOV AX,WORD_INTEGER
    IDIV BYTE_INTEGER        ;QUOTIENT IN AL

(2) MOV CL,DIVISOR_BYTE
    MOV AL,DIVIDEND_BYTE
    CBW
    IDIV CL                  ;QUOTIENT IN AL

(3) MOV CX,DIVISOR_WORD
    MOV AX,DIVIDEND_WORD
    CWD
    IDIV CX                  ;QUOTIENT IN AX
```

Die später beschriebenen Befehle CBW und CWD können benutzt werden, um durch Vorzeichenverlängerung vorzeichenbehaftete 8- Bit- in 16-Bit-Zahlen bzw. 16-Bit- in 32-Bit-Zahlen zu konvertieren.

AAD

Korrigiere den Dividenden vor einer Division mit ungepackten BCD-Zahlen

Dieser Befehl ermöglicht die Division zweier ungepackter Dezimalzahlen, so daß das Ergebnis ebenfalls im ungepackten BCD-Format abgespeichert wird. Dazu muß (im Gegensatz zu den Befehlen AAA, AAS und AAM) der Operand im Akkumulator vor der Division korrigiert werden. Dabei wird angenommen, daß der Akkumulator AX den Dividenden als zweistellige ungepackte BCD-Zahl enthält (Einer in AL, Zehner in AH). Der AAD-Befehl bildet daraus ein binäres Äquivalent. Nach der Division mit dem DIV-Befehl sind dann der

Quotient in AL und der Rest in AH ungepackte Dezimalzahlen.

Beispiel:
```
;DIVIDIERE 41:7
MOV BL,7     ;DIVISOR
MOV AX,401H  ;ENTSPRICHT 41 IM UNGEPACKTEN BCD-KODE
AAD
DIV BL       ;IN AL STEHT 5 ALS QUOTIENT
             ;IN AH STEHT 6 ALS REST
```

CBW

Konvertiere ein Byte in ein Wort

Der Befehl CBW produziert aus einem Byte im AL-Register ein Wort, indem das Vorzeichenbit (Bit 7 in AL) in alle Bits des Registers AH übernommen wird. Für positive Zahlen ist also AH = 0H, für negative wird AH = FFH. Dadurch wird eine als vorzeichenbehaftet angenommene 8-Bit-Zahl in eine wertgleiche 16-Bit-Zahl umgewandelt.

CWD

Konvertiere ein Wort in ein Doppelwort

Der Befehl CWD produziert aus einem Wort im AX-Register ein Doppelwort, indem das Vorzeichenbit (Bit 15 in AX) in alle Bits des Registers DX übernommen wird. Für positive Zahlen ist also DX = 0H, für negative wird DX = FFFFH. Dadurch wird eine als vorzeichenbehaftet angenommene 16-Bit-Zahl in eine wertgleiche 32-Bit-Zahl umgewandelt.

4.4. Befehle zur Bitmanipulation

Diese Gruppe umfaßt zwei Befehlsklassen, die logischen Operationen und die Bitverschiebeoperationen.

4.4.1. Flagbeeinflussung

Die Bitmanipulationsbefehle beeinflussen verschiedene Flags:

O	Überlaufflag	(Overflow)
S	Vorzeichenflag	(Sign)
Z	Nullflag	(Zero)
P	Paritätsflag	(Parity)
C	Übertragsflag	(Carry)

Das Hilfsübertragsflag A ist in der Regel nach logischen Operationen undefiniert. Tafel 4.2 zeigt, welche Flags durch die einzelnen Befehle dieser Gruppe beeinflußt werden.

Tafel 4.2. Beeinflussung der Flags bei Logikoperationen

Befehl		O	S	Z	A	P	C
Boolesche Operationen:	AND, TEST, OR, XOR	O	X	X	U	X	O
	NOT	-	-	-	-	-	-
Verschieben:	SHL/SAL, SHR, SAR	X	X	X	U	X	X
Rotieren:	ROL, ROR, RCL, RCR	X	-	-	-	-	X

X : Flag wird aktualisiert
- : Flag bleibt unbeeinflußt
U : Flag nimmt undefinierten Zustand ein
O : Flag wird auf Null gesetzt

Für die Aktualisierung der Flags durch Bitmanipulationsbefehle gelten folgende Regeln:

1. C enthält stets den Wert des letzten Bits, welches aus dem Zieloperanden herausgeschoben oder -rotiert wurde.

2. O wird auf 1 gesetzt, wenn das höchstwertige Bit des Zieloperanden (Vorzeichenbit) beim Verschieben oder Rotieren um eine Bitposition (count = 1) seinen Wert ändert. Beim Verschieben oder Rotieren um mehrere Positionen nimmt es einen undefinierten Wert ein.

3. Z wird auf 1 gesetzt, wenn das Ergebnis der Operation Null ist.

4. S enthält stets den Wert des höchstwertigen Bits (Vorzeichenbit) des Ergebnisses.

5. P wird auf 1 gesetzt, falls das Ergebnis eine gerade Anzahl von Eins-Bits enthält (gerade Parität). Andernfalls wird es auf 0 gesetzt (ungerade Parität).

Alle Flags und Flagkombinationen können mit bedingten Sprungbefehlen getestet werden (siehe Abschn. 4.6).

4.4.2. Befehle für logische Operationen

Diese Klasse von Befehlen erlaubt die Ausführung logischer (Boolescher) Operationen:

- "NICHT" (NOT),
- "UND" (AND),
- "ODER" (OR),
- "EXKLUSIV-ODER" (XOR).

Ein spezieller TEST-Befehl führt eine logische UND-Operation aus, ohne das Ergebnis abzuspeichern (ähnlich dem arithmetischen CMP-Befehl).
Die logischen Operationen können sowohl Speicher- und Registeroperanden als auch unmittelbare Daten verarbeiten.

NOT dest

Führe eine logische NICHT-Operation aus

Der Befehl NOT bildet das Einerkomplement eines Register- oder Speicheroperanden. Das bedeutet, daß jedes Bit invertiert wird.

AND dest,src

Führe eine logische UND-Operation aus

Dieser Befehl verknüpft die beiden Operanden (Bytes oder Worte) durch eine bitweise UND-Operation. Im Ergebnis werden alle die Bits auf 1 gesetzt, deren entsprechende Bits in beiden Operanden ebenfalls 1 sind. Ist wenigstens eines der beiden Bits 0, so wird auch das entsprechende Ergebnisbit auf 0 gesetzt.

Beispiele:

```
AND BL,BH                ;REGISTER 'UND' REGISTER
AND DI,DELTA             ;REGISTER 'UND' SPEICHER
AND ALPHA[DI],SI         ;SPEICHER 'UND' REGISTER
AND AL,01010100B         ;REGISTER 'UND' KONSTANTE
AND WORD PTR [BX],1FFH   ;SPEICHER 'UND' KONSTANTE
```

Diese Beispiele gelten sinngemäß auch für die folgenden Befehle.

TEST dest,src

Führe einen logischen Vergleich durch

Der Befehl TEST dient dem logischen Vergleichen zweier Operanden. Er arbeitet analog zum Befehl AND mit dem Unterschied, daß die Operanden nicht beeinflußt werden. Das Ergebnis der logischen UND-Verknüpfung wird nicht abgespeichert, sondern nur die Flags aktualisiert. Mit nachfolgenden bedingten Sprungbefehlen kann das Vergleichsergebnis ausgewertet werden.

OR dest,src

Führe eine logische ODER-Operation aus

Dieser Befehl verknüpft die beiden Operanden (Bytes oder Worte) durch eine bitweise ODER-Operation. Im Ergebnis wird ein Bit auf 1 gesetzt, wenn wenigstens eines der beiden korrespondierenden Bits 1 ist. Sind beide Bits 0, so wird auch das entsprechende Bit im Ergebnis auf 0 gesetzt.

XOR dest,src

Führe eine logische Exklusiv-ODER-Operation aus

Dieser Befehl verknüpft die beiden Operanden (Bytes oder Worte) durch eine bitweise Exklusiv-ODER-Operation (Antivalenz). Im Ergebnis wird ein Bit auf 0 gesetzt, wenn beide korrespondierenden Bits den gleichen Wert haben. Sind beide Bits verschieden, so wird das entsprechende Bit im Ergebnis auf 1 gesetzt.

4.4.3. Bitverschiebe- und Rotationsbefehle

Die Befehle dieser Klasse erlauben das Verschieben von Bits in Speicher- oder Registeroperanden. Bei den Schiebebefehlen (Shift) gehen die Bits, die aus dem Operanden herausgeschoben werden, verloren. Die Rotationsbefehle (Rotate) verschieben die Bits im Kreise. Bild 4.2 zeigt die prinzipielle Wirkungsweise der Verschiebebefehle am Beispiel von Byteoperationen. (Wortoperationen verlaufen analog.) Jedes Bit kann um maximal 255 Bitpositionen verschoben werden, wobei die Anzahl der Positionen im CL-Register angegeben wird. Dadurch ist es möglich, daß diese Zahl für einen konkreten Befehl im Programm variabel sein kann.

SHL / SAL dest,count

Verschiebe logisch/arithmetisch nach links

Diese beiden Befehle führen eine identische Operation aus, das heißt, es handelt sich auf Maschinenebene nur um einen Befehl. Das logische Verschieben dient dem Isolieren einzelner Bits, während das arithmetische Linksverschieben zur Multiplikation mit Zweierpotenzen angewendet werden kann. (Es ist jedoch - im Gegensatz zum Rechtsverschieben - belanglos, welche Mnemonik hier angewendet wird.)
Durch diesen Befehl werden die Bits im Zieloperanden (Byte oder Wort) nach links geschoben, also in Richtung zum höchstwertigen Bit. An der niederwertigsten Position (das heißt von rechts) werden Nullen eingeschoben. Das jeweils höchstwertige Bit wird in das Übertragsflag C geschoben. Der Operand "count" gibt an, ob nur eine Verschiebung um eine Stelle oder um die in CL enthaltene Anzahl ausgeführt werden soll.

Beispiele: (1)
```
MOV CL,4
SAL WERT,CL
```

Die in der Variablen WERT enthaltene Zahl wird mit einer Zweierpotenz multipliziert, wobei CL den Exponenten enthält. In diesem Beispiel wird die Zahl also mit 16 multipliziert.

(2)
```
SHL BX,1
```

Der Inhalt von BX wird um eine Stelle verschoben (mit 2 multipliziert); Bit 15 wird nach C transportiert.

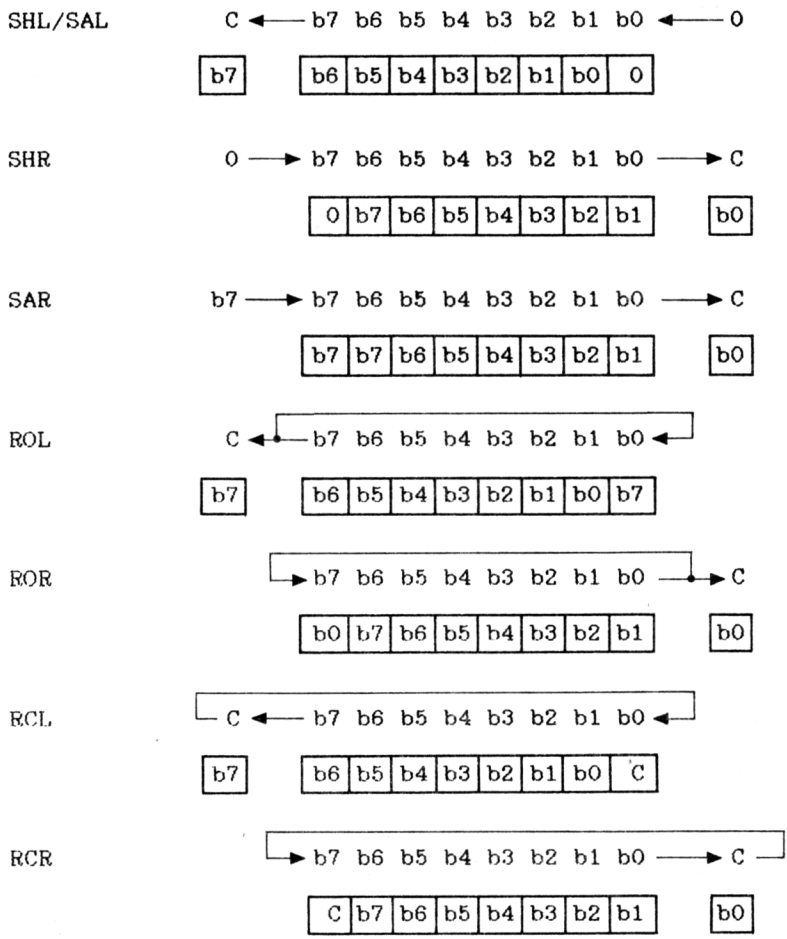


Bild 4.2. Wirkungsweise der Bitverschiebebefehle

SHR dest, count

Verschiebe logisch nach rechts

Durch diesen Befehl werden die Bits im Zieloperanden (Byte oder Wort) nach rechts geschoben, also in Richtung des niederwertigsten Bits. An der höchstwertigen Position (das heißt von links) werden Nullen eingeschoben. Das jeweils niederwertigste Bit wird in das Übertragsflag C geschoben. Der Operand "count" gibt an, ob nur eine Verschiebung um eine Stelle oder um die in CL enthaltene Anzahl ausgeführt werden soll.

SAR dest,count

Verschiebe arithmetisch nach rechts

Dieser Befehl wirkt analog dem logischen Verschieben mit SHR. Die Bits im Zieloperanden (Byte oder Wort) werden nach rechts geschoben, also in Richtung des niederwertigsten Bits. Das jeweils niederwertigste Bit wird in das Übertragsflag C geschoben. Der Unterschied zu SHR besteht darin, daß das höchstwertige Bit seinen Wert behält. Anders ausgedrückt, es werden von links Nullen eingeschoben, wenn der Operand ein positives Vorzeichen hat, und Einsen bei negativem Vorzeichen. Dadurch erlaubt dieser Befehl eine Division von Binärzahlen durch Zweierpotenzen. Der Operand "count" gibt an, ob nur eine Verschiebung um eine Stelle oder um die in CL enthaltene Anzahl ausgeführt werden soll.

Beispiel:

```
MOV CL,POWER ;EXPONENT LADEN
SAR AX,CL    ;DIVISION DURCH 2 HOCH POWER
```

Bei dieser Division ist zu beachten, daß dieser Befehl nicht immer das gleiche Ergebnis liefert wie der Befehl IDIV. Während bei der Integerdivision immer in Richtung Null gerundet wird, rundet das Rechtsschieben zur nächstniederen ganzen Zahl. Beispielsweise liefert die Division -7:2 das Ergebnis -3 mit IDIV und -4 mit SAR.

ROL dest,count

Rotiere nach links

Der ROL-Befehl rotiert die Bits des Zieloperanden linksherum, das heißt in Richtung zum höchstwertigen Bit. Die niederwertigste Bitposition wird mit dem jeweils herausgeschobenen höchstwertigen Bit aufgefüllt. Außerdem wird das herausgeschobene Bit in das Übertragsflag übernommen. Der Operand "count" gibt an, ob nur eine Verschiebung um eine Stelle oder um die in CL enthaltene Anzahl ausgeführt werden soll.

Beispiele:

```
ROL BL,CL
ROL WORD_VARIABLE,1
```

ROR dest,count

Rotiere nach rechts

Der ROR-Befehl rotiert die Bits des Zieloperanden rechtsherum, das heißt in Richtung zum niederwertigsten Bit. Die höchstwertige Bitposition wird mit dem jeweils herausgeschobenen niederwertigsten Bit aufgefüllt. Außerdem wird das herausgeschobene Bit in das Übertragsflag übernommen. Der Operand "count" gibt an, ob nur eine Verschiebung um eine Stelle oder um die in CL enthaltene Anzahl ausgeführt werden soll.

RCL dest,count
Rotiere über das Übertragsflag nach links

Der RCL-Befehl wirkt ähnlich wie der ROL-Befehl, jedoch wird das Übertragsflag als zusätzliche Bitposition in die Schiebekette mit einbezogen. Alle Bits werden nach links, das heißt in Richtung zum höchstwertigen Bit verschoben. Das höchstwertige Bit wird aus dem Zieloperanden heraus in das Übertragsflag geschoben. Die niederwertigste Bitposition wird mit dem alten Wert des Übertragsflags aufgefüllt. Der Operand "count" gibt an, ob nur eine Verschiebung um eine Stelle oder um die in CL enthaltene Anzahl ausgeführt werden soll.

RCR dest,count
Rotiere über das Übertragsflag nach rechts

Der RCR-Befehl wirkt ähnlich wie der ROR-Befehl, jedoch wird das Übertragsflag als zusätzliche Bitposition in die Schiebekette mit einbezogen (analog zum RCL-Befehl). Alle Bits werden nach rechts, das heißt in Richtung zum niederwertigsten Bit verschoben. Das niederwertigste Bit wird aus dem Zieloperanden heraus in das Übertragsflag geschoben. Die höchstwertige Bitposition wird mit dem alten Wert des Übertragsflags aufgefüllt. Der Operand "count" gibt an, ob nur eine Verschiebung um eine Stelle oder um die in CL enthaltene Anzahl ausgeführt werden soll.

4.5. Stringbefehle

Die Befehle dieser Gruppe ermöglichen die Manipulierung von Zeichenketten (Strings) bis zu 64 KByte Länge. Die fünf Basisoperationen realisieren Übertragungs-, Vergleichs- und Suchoperationen mit jeweils einem Stringelement. Ein Element kann vom Typ Byte oder Wort sein. Die Verbindung eines Grundbefehls mit einem Repeat-Präfix veranlaßt die Prozessorhardware zur Wiederholung der Operation, um Strings beliebiger Länge bearbeiten zu können. Diese Methode spart Speicherplatz und arbeitet wesentlich schneller als eine Softwareschleife. Da Repeat-Präfixe nur bei Stringbefehlen angewendet werden, sollen sie in diesem Abschnitt mit behandelt werden.

4.5.1. Adressierung, Register- und Flagbenutzung

Alle Stringbefehle sind durch einige gemeinsame Merkmale gekennzeichnet, die den speziellen Befehlsbeschreibungen vorangestellt werden sollen.
Die Adressierung erfolgt prinzipiell indirekt über die Register DI und SI.
Der K1810WM86 nimmt an, daß der Quellstring im aktuellen Datensegment (DS) lokalisiert ist und das Register SI den Offset des Elements enthält. Ein Segmentoverride-Präfix kann die Segmentzu-

ordnung aufheben. Der Zielstring muß im aktuellen Extradatensegment (ES) liegen; die Offsetadresse ist in DI enthalten. (Ein Segmentoverride ist hier nicht möglich, da nur ein Override-Präfix je Befehl generiert werden kann.) Vor einem Stringbefehl müssen diese Register entsprechend initialisiert sein. Die Tafel 4.3 zeigt die Adressierung der Stringoperanden im Überblick.

Tafel 4.3. Adressierung der Stringelemente

Befehl	Zieloperand	Quelloperand
MOVS	ES : DI	DS (ES,SS,CS) : SI
CMPS	ES : DI	DS (ES,SS,CS) : SI
SCAS	ES : DI	AX / AL
LODS	AX / AL	DS (ES,SS,CS) : SI
STOS	ES : DI	AX / AL

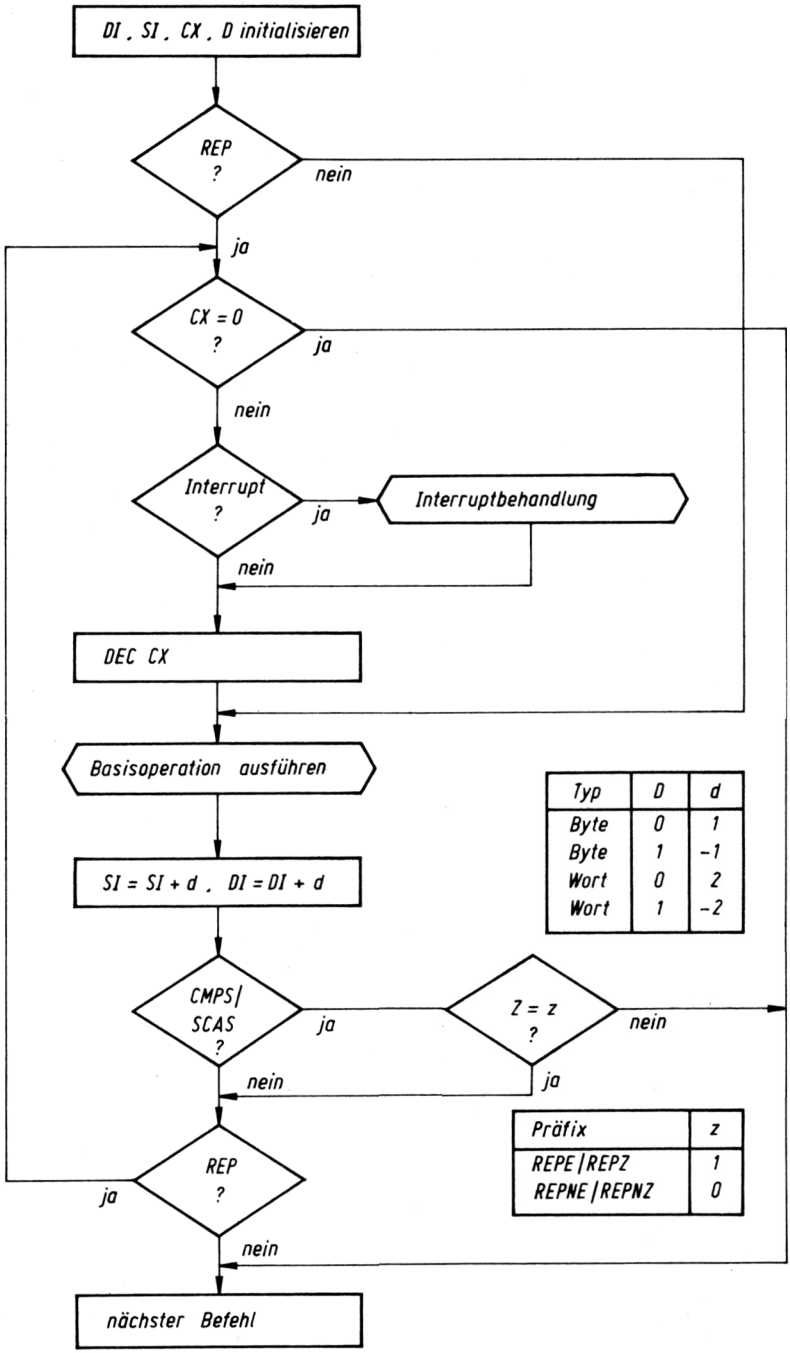
Da bei Stringbefehlen keine Adreßrechnung wie bei anderen Speicheroperationen erforderlich ist, müssen keine Operanden angegeben werden. Der Assembler muß aber den Typ der Operation kennen (Byte oder Wort). Dieser kann in der Mnemonik angegeben werden. Eine andere mnemonische Form verwendet Operandenangaben. Diese ist sinnvoll, wenn Quell- und Zielstring als Variable vereinbart sind. Der Assembler ermittelt hierbei den Operationstyp aus der Variablenvereinbarung (Byte oder Wort). Eine Operandenangabe ist auch bei Segmentüberschreibung erforderlich.

Beispiele:

```
MOVSB                          ;BYTETRANSFER
MOVSW                          ;WORTTRANSFER
MOVS DEST_STRING,SRC_STRING    ;BYTE ODER WORT
MOVS [DI],CS:WORD PTR [SI]     ;WORTTRANSFER MIT
                               ;SEGMENTOVERRIDE
```

Für die Initialisierung der Register können neben den MOV-Befehlen auch LDS und LES angewendet werden.

Die Stringbefehle aktualisieren automatisch nach jeder Einzeloperation den Index. In Abhängigkeit vom Richtungsflag D werden SI und DI erhöht (D=1) oder erniedrigt (D=0), und zwar um den Wert 1 bei Byte- oder 2 bei Wortoperanden. D muß ebenfalls vor der Stringoperation initialisiert sein; die Befehle STD und CLD sind dafür vorgesehen. Nach jeder Basisoperation zeigen die Indexregister also auf das nächste Stringelement.
Wird ein Repeat-Präfix angewendet, so muß das Register CX mit der Anzahl der Stringelemente geladen werden. Bei jeder Basisoperation wird CX um den Wert 1 erniedrigt. Die Basisoperation wird dann so oft wiederholt, bis CX=0 ist. Bei den Such- und Vergleichsbefehlen CMPS und SCAS kann außerdem der Zustand des Nullflags Z als Endebedingung für wiederholte Stringoperationen vorgegeben werden. Bild 4.3 zeigt den prozessorinternen Ablauf bei der Abarbeitung eines Stringbefehls. Die Tafel 4.4 zeigt die speziellen Funktionen der Register und Flags.



Typ	D	d
Byte	0	1
Byte	1	-1
Wort	0	2
Wort	1	-2

Präfix	z
REPE/REPZ	1
REPNE/REPNZ	0

Bild 4.3. Prozessorinterner Ablauf bei Stringoperationen

Tafel 4.4. Anwendung der Register und Flags bei Stringbefehlen

Register	Anwendung (implizit)
SI	Index (Offset) des Quellelements
DI	Index (Offset) des Zielelements
DS	Basisadresse des Quellstrings (falls kein Segmentoverride-Präfix angegeben)
ES	Basisadresse des Zielstrings
CX	Wiederholungszähler für Repeat-Präfix
AX / AL	Zieloperand bei LODS, Quelloperand bei STOS, Quelloperand (Vergleichswert) bei SCAS
Flag	**Anwendung (implizit)**
D	0 : SI / DI erhöhen 1 : SI / DI erniedrigen
Z	0 : Wiederholung beenden bei REPNE/REPNZ 1 : Wiederholung beenden bei REPE/REPZ

4.5.2. Repeat-Präfix

Das Repeat-Präfix kann ausschließlich in Verbindung mit Stringbefehlen angewendet werden. Es veranlaßt den K1810WM86, eine Grundoperation so oft zu wiederholen, bis eine Endebedingung erreicht ist. Tafel 4.5 zeigt alle Formen dieses Präfixes und ihre Anwendung.

Wiederholte Stringoperationen sind unterbrechbar, das heißt, nach jeder Basisoperation (nach jedem Byte- oder Worttransfer) werden die Signale INTR und NMI getestet. Bei Vorliegen einer gültigen Anforderung wird zur Interruptprozedur gesprungen. Nach der Rückkehr kann die Stringoperation an der Stelle fortgesetzt werden, an der sie unterbrochen wurde.
Sollen mehrere Präfixe angewendet werden, so ist bei auftretenden Unterbrechungen keine korrekte Arbeitsweise des Stringbefehls mehr möglich. Der Grund hierfür liegt darin, daß der Prozessor sich nach der Rückkehr nur noch an das unmittelbar vor dem Befehlskode stehende Präfix "erinnert". Soll beispielsweise ein Befehl in der Form

```
LOCK REP MOVS DEST_STRING,CS:SOURCE_STRING
```

ausgeführt werden, so müssen Interrupts gesperrt werden. Das wird jedoch nicht immer günstig sein, zumal ein nichtmaskierbarer Interrupt immer auftreten kann. Außerdem können Programme mit Befehlen dieser Form schlecht getestet werden, da hierbei meist der Einzelschrittbetrieb durch Setzen des Trap-Flags anwendbar sein soll, der ebenfalls eine Unterbrechung bewirkt.

Tafel 4.5. Anwendung des Repeat-Präfixes

Mnemonik	Anwendung bei	Endebedingung
REP	MOVS, STOS	CX = 0
REPE/REPZ	CMPS, SCAS	CX = 0 oder Z = 0
REPNE/REPNZ	CMPS, SCAS	CX = 0 oder Z = 1

REP / REPE / REPZ
Wiederhole / Wiederhole, falls das Nullflag gesetzt ist

Diese drei Präfixe sind physikalisch identisch; es wird das gleiche Präfixbyte für alle drei Mnemoniks generiert. Das Präfix REP wird in Verbindung mit den Befehlen MOVS und STOS angewendet und als "Wiederhole bis String-Ende" interpretiert. Vor der Operation muß das CX-Register mit der Zahl der Stringelemente geladen werden. Während jeder Basisoperation wird dann CX um den Wert 1 erniedrigt und die Operation so oft wiederholt, solange CX von Null verschieden ist.
Die Präfixe REPE oder REPZ können bei den Befehlen CMPS und SCAS angewendet werden. (Dabei ist es belanglos, welche der beiden Mnemoniks verwendet werden.) Diese Präfixe verlangen, daß sowohl CX von Null verschieden als auch das Nullflag Z auf 1 gesetzt ist, um die Basisoperation zu wiederholen.

REPNE / REPNZ
Wiederhole, falls das Nullflag nicht gesetzt ist

Diese beiden physikalisch identischen Präfixe wirken ähnlich wie REPE/REPZ. Der Unterschied besteht darin, daß das Nullflag Z auf 0 gesetzt sein muß, um eine Wiederholung der Operation zu bewirken. Sie werden ebenfalls nur bei den Befehlen SCAS und COMPS angewendet.

4.5.3. Basisoperationen

Zur Bearbeitung der Strings können fünf verschiedene Grundbefehle angewendet werden. Die Basisoperationen können in Softwareschleifen angewendet werden; effektiver ist jedoch die Verbindung mit einem Repeat-Präfix, um Blockoperationen auszuführen.

MOVS dest,src / MOVSB / MOVSW
Übertrage einen Byte- oder Wortstring

Dieser Befehl transportiert ein Byte oder Wort aus dem Quellstring zum Zielstring. Im Gegensatz zum normalen MOV-Befehl ist ein Speicher-Speicher-Transfer möglich. In Verbindung mit dem Präfix REP können Blockübertragungen mit bis zu 64 KByte großen Datenblöcken realisiert werden.

Beispiele: (1)
```
MOV SI, OFFSET SOURCE_STRING
MOV DI, OFFSET DEST_STRING
MOV CX, LENGTH SOURCE_STRING
REP MOVS DEST_STRING, SOURCE_STRING
```

Dieses Beispiel einer Blockübertragung setzt voraus, daß DS und ES initialisiert sind und die Variablen DEST_STRING und SOURCE_STRING vereinbart sind. Dann ist diese Prozedur geeignet, beliebige Blöcke zu übertragen.

(2)
```
LDS SI,SOURCE_POINTER
LES DI,DEST_POINTER
MOVSB
```

Dieses Beispiel erfordert keine Variablenvereinbarung für die Strings, jedoch müssen zwei Doppelwortzeiger vereinbart werden, welche die absoluten Adressen von Quell- und Zielstring enthalten.

STOS dest / STOSB / STOSW

Speichere einen Byte- oder Wortstring

Dieser Befehl transportiert ein Byte oder Wort aus dem Akkumulator zum Zielstring. In Verbindung mit dem Präfix REP kann dieser Befehl benutzt werden, um einen String bis zu 64 KByte Länge mit einem konstanten Wert zu initialisieren.

Beispiel:
```
LES DI, DISP_LINE
MOV AL, ' '
REP STOSB
```

Eine solche Prozedur kann benutzt werden, um etwa eine Bildschirmzeile mit gleichen Zeichen (z.B. Leerzeichen) zu füllen. Der Pointer DISP_LINE enthält die Adresse der Zeile im Bildspeicher, AL wird mit dem Zeichen geladen. CX muß vorher mit der Zeilenlänge initialisiert werden.

LODS src / LODSB / LODSW

Lade einen Byte- oder Wortstring

Dieser Befehl transportiert ein Byte oder Wort aus dem Quellstring zum Akkumulator.

Beispiel:
```
LODS ASCII_BUFFER
```

Dieser Befehl kann nicht sinnvoll mit einem Repeat-Präfix verbunden werden, da dann der Akkumulatorinhalt ständig überschrieben wird und nur das letzte Element erhalten bleibt. Die Anwendung ist jedoch günstig in Programmschleifen zur komplexen Zeichenkettenbearbeitung.

CMPS dest,src / CMPSB / CMPSW

Vergleiche Byte- oder Wortstrings

Dieser Befehl funktioniert ähnlich wie CMP (s. Abschn. 4.3.4). Er subtrahiert zwei Stringelemente voneinander und aktualisiert die Flags in Abhängigkeit vom Ergebnis. Die Operanden selbst werden nicht beeinflußt.
Das Vergleichsergebnis kann auf verschiedenen Wegen ausgetestet werden. In Softwareschleifen kann das Ergebnis jeder Basisoperation durch nachfolgende bedingte Sprungbefehle ausgewertet werden. Will man Strings beliebiger Länge (maximal 64 KByte) auf gleiche oder abweichende Elemente durchsuchen, kann mit den Präfixen REPE/REPZ oder REPNE/REPNZ gearbeitet werden.

Beispiel:

```
MOV SI, END_STRING1
MOV DI, END_STRING2
MOV CX, LENGTH
STD
REPE CMPSW
```

Diese Befehlsfolge initialisiert SI und DI mit den Endeadressen zweier Strings. Diese werden vom Ende beginnend verglichen. Bei Gleichheit setzt der CMPS-Befehl Z auf 1 und der Vergleich wird fortgesetzt. Er wird abgebrochen, wenn sich zwei Elemente unterscheiden oder die in CX enthaltene Anzahl von Elementen durchgesehen wurde. Anhand der Offsetadressen in SI bzw. DI können die entsprechenden Elemente ermittelt werden.

SCAS dest / SCASB / SCASW

Teste einen Byte- oder Wortstring

Der Befehl SCAS dient dem Durchsuchen eines Strings nach einem im Akkumulator enthaltenen Zeichen. Das Stringelement wird vom Akkumulatorinhalt subtrahiert und die Flags werden aktualisiert. Die Operanden selbst werden nicht beeinflußt.
Das Vergleichsergebnis kann auf verschiedenen Wegen ausgetestet werden. In Softwareschleifen kann das Ergebnis jeder Basisoperation durch nachfolgende bedingte Sprungbefehle ausgewertet werden. Will man Strings beliebiger Länge (maximal 64 KByte) auf das Vorhandensein oder Fehlen eines bestimmten Elements prüfen, kann mit den Präfixen REPE/REPZ oder REPNE/REPNZ gearbeitet werden.

Beispiel:

```
MOV AL, TEST_BYTE
MOV DI, OFFSET STRING
MOV CX, LENGTH STRING
CLD
REPNE SCAS STRING
```

Diese Befehlsfolge durchsucht einen String nach einem bestimmten Datenbyte. Der String wurde als Folge von Bytevariablen vereinbart. Bei Gleichheit wird Z auf 1 gesetzt und der Vergleich abgebrochen.

4.6. Programmtransferbefehle

Die Gruppe der Programmtransferbefehle ermöglicht die Programmierung von Sprüngen, Schleifen und den Aufruf von Unterprogrammen.

4.6.1. Wirkungsweise und Operanden

In einem K1810WM86-Programm ist jeder Befehl durch eine Kodesegmentadresse (Inhalt von CS) und eine Offsetadresse (Inhalt von IP) im Speicherraum lokalisiert. Normalerweise arbeitet der K1810WM86 ein Programm linear ab, indem ständig IP erhöht wird und Befehle von fortlaufenden Speicheradressen eingelesen werden. Da der Prozessor sechs Befehlsbytes in einem internen RAM, der Befehlswarteschlange, abspeichern kann, ist der Befehl, der als nächster ausgeführt wird, meist schon eingelesen (siehe auch Abschn. 2.1.4). IP zeigt dann auf einen der nachfolgenden Befehle. Ein Programmtransferbefehl bewirkt nun folgende Operationen:

1. Die Befehlswarteschlange wird gelöscht. IP wird auf den Befehl justiert, der im Speicher auf den Programmtransferbefehl folgt.
2. IP und gegebenenfalls CS werden mit der Adresse des Zielbefehls geladen.
3. Der Zielbefehl wird eingelesen und ausgeführt.
4. Die Befehlswarteschlange wird fortlaufend gefüllt.

Die Operanden müssen bei diesen Befehlen also Adreßinformationen sein. Bei der direkten Form ist die Zieladresse als unmittelbarer Operand im Befehlskode enthalten. Sie wird vom Assembler aus einer Marke (Label) ermittelt, die als Operand im Assemblerbefehl verwendet wird und auf den Zielbefehl weist.
Erst wenn das Quellprogramm in ein absolutes Objektmodul, das heißt in ein ausführbares Maschinenprogramm übersetzt wird, werden die Marken durch die unmittelbaren Operanden ersetzt.
Bei der indirekten Form ist die Zieladresse in einem Zeiger (Pointer) enthalten. Das kann ein 16-Bit-Register oder eine Speicherzelle (16 oder 32 Bit) sein. Diese werden wie Variable behandelt.

Die Operanden werden in drei Typen eingeteilt:

- Typ "far" (fern) oder Intersegment:
  Die Zieladresse befindet sich außerhalb des aktuellen Kodesegments.
  Der Operand ist ein Doppelwort (32 Bit). Er kann unmittelbar im Befehl enthalten sein oder in einer Doppelwortvariablen im Speicher. Das niederwertige Wort (Offsetwort) ersetzt IP; das höherwertige (Kodesegmentwort) ersetzt CS.

  Dieser Typ kann nur bei unbedingten Transfers (CALL, JMP) angewendet werden.

- Typ "near" (nah) oder Intrasegment:
  Die Zieladresse befindet sich innerhalb des aktuellen Kodesegments.
  Der Operand ist ein Wort. Beim direkten Transfer ist er im Befehl enthalten und wird als vorzeichenbehaftete Zahl interpretiert, die zum aktuellen Inhalt von IP addiert wird. Der Operand gibt also den Abstand des Zielbefehls vom aktuellen Befehl im Bereich ±32 KByte an.
  Beim indirekten Transfer ist das Wort in einem Register oder einer Speichervariablen enthalten. Es ersetzt den Inhalt von IP.

- Typ "short" (kurz) oder Intrasegment-kurz:
  Der Operand ist ein Byte, welches unmittelbar im Befehl enthalten ist. Es wird als vorzeichenbehaftete Zahl interpretiert und zu IP addiert. Es gibt den Abstand des Zielbefehls vom aktuellen Befehl im Bereich -128 bis +127 Bytes an. Dieser Typ kann nur in der direkten Form angewendet werden. Bedingte Sprungbefehle und Schleifenbefehle können nur Operanden vom Typ "short" verwenden.

Direkte Intersegmenttransfers werden vom Assembler generiert, wenn der Operand eine Marke (oder ein Prozedurname) ist, die als "far" vereinbart wurde. Ist der Operand als Doppelwortvariable vereinbart, wird ein indirekter Intersegmenttransfer ausgeführt.

Der Assembler generiert direkte Intrasegmenttransfers, wenn der Operand eine Marke (oder ein Prozedurname) ist, die als "near" vereinbart wurde. Nicht vereinbarte Marken werden als "near" interpretiert. Befindet sich die Zieladresse im Bereich von -128 bis +127 Bytes, so wird automatisch der Typ "short" angewendet. Ist der Operand ein 16-Bit-Register oder eine als Wort vereinbarte Variable, so wird ein indirekter Intrasegmenttransfer ausgeführt. Direkte Intrasegmenttransfers können vorteilhaft für dynamisch verschiebliche Programme verwendet werden, da der Abstand zwischen zwei Befehlen in einem Segment erhalten bleibt.

4.6.2. Prozeduraufrufe

Ein Prozeduraufruf veranlaßt den K1810WM86, in ein Unterprogramm zu springen, dieses abzuarbeiten und anschließend wieder in das aufrufende Programm zurückzukehren. Der prinzipielle Ablauf ist dabei folgender:

1. Ein CALL-Befehl überträgt die Adresse des nachfolgenden Befehls in den Stapel und lädt die Adresse des ersten Befehls der Prozedur in IP. (Bei Prozeduren vom Typ "far" wird auch CS geladen.)
2. Die Prozedur wird abgearbeitet, bis sie durch einen Rückkehrbefehl RET beendet wird. Dieser lädt aus dem Stapel die Adresse des Befehls, der vom CALL-Befehl dort abgelegt wurde.
3. Das aufrufende Programm wird fortgesetzt.

Es sind beliebige Schachtelungstiefen von Unterprogrammen möglich; Grenzen setzt nur die Größe des Stapels.
Prozeduren können vom Typ "far" oder "near" sein. Der Typ muß durch eine Assemblerdirektive deklariert sein. (Nichtdeklarierte Prozeduren werden als "near" interpretiert.) Werden Prozedur und aufrufendes Programm unabhängig voneinander assembliert, so ist darauf zu achten, daß der Typ einheitlich vereinbart wird. Damit wird gesichert, daß der Rückkehrbefehl in der Prozedur vom gleichen Typ ist wie der Aufruf.

CALL procedure

Rufe eine Prozedur auf

Der Befehl CALL bewirkt einen unbedingten Sprung zu einem Zielbefehl, dessen Speicheradresse aus dem Operanden ermittelt wird. Der Operand kann ein Prozedurname oder ein Pointer (Register oder Speichervariable) sein.
Der Intrasegment-CALL ("Near CALL") überträgt den Inhalt des Befehlszeigers IP in den Stapel und lädt die Offsetadresse des Zielbefehls. Beim Intersegment-CALL ("Far CALL") wird zuerst der Inhalt des Kodesegmentregisters CS im Stapel abgelegt und die neue Kodesegmentadresse geladen, anschließend wird der IP-Inhalt im Stapel abgelegt und die Offsetadresse geladen.

```
Beispiele: (1)  CALL NEAR_LABEL          ;DIREKT INTRASEGMENT
```

Bei diesem Typ wird beim Übersetzen des Quellprogramms der Abstand der Marke NEAR LABEL von der Offsetadresse des CALL-Befehls ermittelt. Im Maschinenbefehl wird dieser Abstand im Bereich ±32 KByte als 16-Bit-Displacement angegeben.

```
           (2)  CALL FAR_PROCEDURE       ;DIREKT INTERSEGMENT
```

Die Marke FAR_PROCEDURE wird durch ihre Kodesegment- und Offsetadresse repräsentiert, die nach dem Binden der Module als Operand des Maschinenbefehls eingesetzt wird.

```
           (3)  CALL BX                  ;INDIREKT INTRASEGMENT
                                         ;UEBER REGISTERPOINTER
           (4)  CALL WORD PTR [DI]       ;INDIREKT INTRASEGMENT
                                         ;UEBER SPEICHERPOINTER
           (5)  CALL DWORD PTR [BX]      ;INDIREKT INTERSEGMENT
```

RET / RET n

Kehre aus einer Prozedur zurück / - und addiere zu SP

Der Befehl RET bewirkt die Rückkehr aus einer Prozedur zu dem Befehl, welcher auf den CALL folgt, der die Prozedur aufgerufen hat. Dabei wird ein unbedingter Sprung zur Rückkehradresse ausgeführt, die aus dem Stapel gelesen wird. (Der CALL hat diese Adresse vorher im Stapel hinterlegt.)

Der RET-Befehl muß dabei dem Typ des CALL-Befehls entsprechen. Ist die Prozedur vom Typ "near", wird ein Intrasegment-RET generiert, bei dem nur der Befehlszeiger IP aus dem Stapel rekonstruiert wird. Ist die Prozedur als "far" deklariert, so wird ein Intersegment-RET ausgeführt. Dabei werden CS und IP aus dem Stapel rekonstruiert.
Wird im RET-Befehl ein unmittelbarer Wert als Operand angegeben, so wird dieser nach dem Auslesen der Rückkehradresse aus dem Stapel zusätzlich zum Stapelzeigerinhalt (SP) addiert. Damit wird die aktuelle Stapelspitze nicht auf das der Rückkehradresse folgende Wort justiert, sondern die angegebene Anzahl von Bytes wird übersprungen. Diese Befehlsform erweist sich als effektiv, wenn das aufrufende Programm vor dem CALL noch eine Anzahl von Parametern für die Prozedur im Stapel bereitstellt.

Beispiel:

```
;PARAMETER WERDEN BEREITGESTELLT
PUSH PARAMETER_1
PUSH PARAMETER_2
CALL NEAR_PROC                ;AUFRUF DER PROZEDUR
;ES FOLGT DER NAECHSTE BEFEHL
;...

NEAR_PROC    PROC   NEAR      ;TYPDEKLARATION
;PROZEDUR PROLOG
PUSH BX                       ;RETTEN DER REGISTER
PUSH BP
PUSHF
;JETZT FOLGT DIE PROZEDUR
;PROZEDUR EPILOG
POPF
POP BP
POP BX
RET 4                         ;DIE PARAMETER WERDEN
                              ;UEBERSPRUNGEN
```

In diesem Beispiel wird der Stapel für verschiedene Zwecke genutzt. Zuerst werden 2 Parameter abgelegt. Der CALL überträgt anschließend IP. Die Prozedur selbst rettet noch 2 Register und die Flags. Da die Parameter nach der Rückkehr nicht mehr benötigt werden, justiert der RET-Befehl den Stapelzeiger so, daß er auf die Adresse über den beiden Parametern zeigt.

4.6.3. Sprungbefehle

Sprungbefehle veranlassen den K1810WM86, die lineare Programmabarbeitung zu verlassen und mit dem Zielbefehl fortzusetzen. Im Gegensatz zu Prozeduraufrufen wird keine Information in den Stapel übertragen und es wird keine Rückkehr an die Stelle nach dem Sprung erwartet. Die Sprungbefehle können in zwei Gruppen eingeteilt werden:

Unbedingte Sprünge:
Der Sprung wird immer ausgeführt, wenn der Befehl JMP abgearbeitet wird. Das Ziel kann vom Typ "far", "near" oder "short" sein.

Bedingte Sprünge:
Der Sprung wird nur ausgeführt, wenn eine vorgegebene Bedingung erfüllt ist. Diese Bedingungen werden über die Flags O, S, Z, P, C ausgetestet. Der Assembler kann 16 verschiedene Maschinenbefehle generieren. Dafür können 30 verschiedene Mnemoniks angewendet werden. (Insbesondere können die Begriffe "größer" und "kleiner" für vorzeichenbehaftete Operanden angewendet werden, für vorzeichenlose Operanden hingegen "über" und "unter".) In Tafel 4.6 sind alle Sprungbedingungen aufgelistet.
Bedingte Sprünge sind immer vom Typ "short", Operand ist eine Marke.

Tafel 4.6. Testbedingungen für bedingte Sprünge

Bedingung	Test	Befehl
gleich	Z=1	JE (JZ)
ungleich	Z=0	JNE (JNZ)
größer (nicht kleiner oder gleich)	(SxorO)orZ=0	JG / JNLE
kleiner (nicht größer oder gleich)	SxorO=1	JL / JNGE
größer oder gleich (nicht kleiner)	SxorO=0	JGE / JNL
kleiner oder gleich (nicht größer)	(SxorO)orZ=1	JLE / JNG
über (nicht unter oder gleich)	CorZ=0	JA / JNBE
unter (nicht über oder gleich)	C=1	JB / JNAE
über oder gleich (nicht unter)	C=0	JAE / JNB
unter oder gleich (nicht über)	CorZ=1	JBE / JNA
Null	Z=1	JZ (JE)
nicht Null	Z=0	JNZ (JNE)
Übertrag	C=1	JC
kein Übertrag	C=0	JNC
Überlauf	O=1	JO
kein Überlauf	O=0	JNO
Vorzeichen negativ	S=1	JS
Vorzeichen positiv	S=0	JNS
Parität (Parität gerade)	P=1	JP
keine Parität (Parität ungerade)	P=0	JNP

or bedeutet logisches ODER
xor bedeutet logisches Exclusiv-ODER

JMP target

Springe zum Zielbefehl

Der Befehl JMP bewirkt einen unbedingten Sprung zu einem Zielbefehl, dessen Speicheradresse aus dem Operanden ermittelt wird. Der Operand kann eine Marke oder ein Pointer (Register oder Speichervariable) sein. Im Unterschied zum CALL-Befehl werden keine Informationen in den Stapel übertragen.
Beim direkten Intrasegment-JMP generiert der Assembler automatisch einen Befehl vom Typ "short", wenn der Zielbefehl einen Abstand im Bereich +127 bis -128 Bytes vom JMP-Befehl hat.

```
Beispiele: (1) JMP START                ;DIREKT INTRASEGMENT
```

Bei diesem Typ wird beim Übersetzen des Quellprogramms der Abstand der Marke START von der Offsetadresse des JMP-Befehls ermittelt. Im Maschinenbefehl wird dieser Abstand als 16-Bit-Displacement angegeben, falls er im Bereich ±32 KByte liegt (near), oder als 8-Bit-Displacement, falls er im Bereich +127 bis -128 Bytes liegt (short).

```
           (2) JMP FAR_LABEL            ;DIREKT INTERSEGMENT
```

Bei diesem Typ wird die Kodesegment- und Offsetadresse beim Binden der Module ermittelt und als unmittelbarer Wert im Maschinenbefehl angegeben.

```
           (3) JMP CX                   ;INDIREKT INTRASEGMENT
                                        ;UEBER REGISTERPOINTER
           (4) JMP WORD PTR [BP]        ;INDIREKT INTRASEGMENT
                                        ;UEBER SPEICHERPOINTER
           (5) JMP FAR_POINTER          ;INDIREKT INTERSEGMENT
```

Im letzten Beispiel muß die Variable FAR POINTER als Doppelwort vereinbart sein.

JE / JZ target

Springe, falls "gleich" / "Null" erfüllt ist

Dieser Befehl bewirkt einen Sprung zu einer Marke im Abstand +127 bis -128 Bytes, falls das Nullflag Z auf 1 gesetzt ist. Andernfalls wird der auf den Sprung folgende Befehl abgearbeitet.

JNE / JNZ target

Springe, falls "ungleich" / "nicht Null" erfüllt ist

Dieser Befehl bewirkt einen Sprung zu einer Marke im Abstand +127 bis -128 Bytes, falls das Nullflag Z auf 0 gesetzt ist. Andernfalls wird der auf den Sprung folgende Befehl abgearbeitet.

JG / JNLE target

Springe, falls "größer" / "nicht kleiner oder gleich" erfüllt ist

Dieser Befehl bewirkt einen Sprung zu einer Marke im Abstand +127 bis -128 Bytes, falls das Nullflag Z auf 0 gesetzt ist und das Vorzeichenflag S und das Überlaufflag O den gleichen Wert haben. Andernfalls wird der auf den Sprung folgende Befehl abgearbeitet.

JL / JNGE target

Springe, falls "kleiner" / "nicht größer oder gleich" erfüllt ist

Dieser Befehl bewirkt einen Sprung zu einer Marke im Abstand +127 bis -128 Bytes, falls das Vorzeichenflag S und das Überlaufflag O voneinander verschieden sind. Andernfalls wird der auf den Sprung folgende Befehl abgearbeitet.

JGE / JNL target

Springe, falls "größer oder gleich" / "nicht kleiner" erfüllt ist

Dieser Befehl bewirkt einen Sprung zu einer Marke im Abstand +127 bis -128 Bytes, falls das Vorzeichenflag S und das Überlaufflag O den gleichen Wert haben. Andernfalls wird der auf den Sprung folgende Befehl abgearbeitet.

JLE / JNG target

Springe, falls "kleiner oder gleich" / "nicht größer" erfüllt ist

Dieser Befehl bewirkt einen Sprung zu einer Marke im Abstand +127 bis -128 Bytes, falls das Nullflag Z auf 1 gesetzt ist oder das Vorzeichenflag S und das Überlaufflag O voneinander verschieden sind. Andernfalls wird der auf den Sprung folgende Befehl abgearbeitet.

JA / JNBE target

Springe, falls "über" / "nicht unter oder gleich" erfüllt ist

Dieser Befehl bewirkt einen Sprung zu einer Marke im Abstand +127 bis -128 Bytes, falls das Nullflag Z und das Übertragsflag C auf 0 gesetzt sind. Andernfalls wird der auf den Sprung folgende Befehl abgearbeitet.

JB / JNAE / JC target

Springe, falls "unter" / "nicht über oder gleich" / "Übertrag" erfüllt ist

Dieser Befehl bewirkt einen Sprung zu einer Marke im Abstand +127 bis -128 Bytes, falls das Übertragsflag C auf 1 gesetzt ist. Andernfalls wird der auf den Sprung folgende Befehl abgearbeitet.

JAE / JNB / JNC target

Springe, falls "über oder gleich" / "nicht unter" / "kein Übertrag" erfüllt ist

Dieser Befehl bewirkt einen Sprung zu einer Marke im Abstand +127 bis -128 Bytes, falls das Übertragsflag C auf 0 gesetzt ist. Andernfalls wird der auf den Sprung folgende Befehl abgearbeitet.

JBE / JNA target

Springe, falls "unter oder gleich" / "nicht über" erfüllt ist

Dieser Befehl bewirkt einen Sprung zu einer Marke im Abstand +127 bis -128 Bytes, falls das Nullflag Z oder das Übertragsflag O (oder beide) auf 1 gesetzt sind. Andernfalls wird der auf den Sprung folgende Befehl abgearbeitet.

JO target

Springe, falls "Überlauf" erfüllt ist

Dieser Befehl bewirkt einen Sprung zu einer Marke im Abstand +127 bis -128 Bytes, falls das Überlaufflag O auf 1 gesetzt ist. Andernfalls wird der auf den Sprung folgende Befehl abgearbeitet.

JNO target

Springe, falls "kein Überlauf" erfüllt ist

Dieser Befehl bewirkt einen Sprung zu einer Marke im Abstand +127 bis -128 Bytes, falls das Überlaufflag O auf 0 gesetzt ist. Andernfalls wird der auf den Sprung folgende Befehl abgearbeitet.

JS target

Springe, falls "Vorzeichen negativ" erfüllt ist

Dieser Befehl bewirkt einen Sprung zu einer Marke im Abstand +127 bis -128 Bytes, falls das Vorzeichenflag S auf 1 (negatives Ergebnis) gesetzt ist. Andernfalls wird der auf den Sprung folgende Befehl abgearbeitet.

JNS target

Springe, falls "Vorzeichen positiv" erfüllt ist

Dieser Befehl bewirkt einen Sprung zu einer Marke im Abstand +127 bis -128 Bytes, falls das Vorzeichenflag S auf 0 (positives Ergebnis) gesetzt ist. Andernfalls wird der auf den Sprung folgende Befehl abgearbeitet.

JP / JPE target

Springe, falls "Parität" / "Parität gerade" erfüllt ist

Dieser Befehl bewirkt einen Sprung zu einer Marke im Abstand +127 bis -128 Bytes, falls das Paritätsflag P auf 1 gesetzt ist. Andernfalls wird der auf den Sprung folgende Befehl abgearbeitet.

JNP / JPO target

Springe, falls "keine Parität" / "Parität ungerade" erfüllt ist

Dieser Befehl bewirkt einen Sprung zu einer Marke im Abstand +127 bis -128 Bytes, falls das Paritätsflag P auf 0 gesetzt ist. Andernfalls wird der auf den Sprung folgende Befehl abgearbeitet.

4.6.4. Befehle zur Schleifensteuerung

Diese Befehle dienen der Steuerung von Programmschleifen. Ähnlich wie beim Repeat-Präfix wird das Register CX als Schleifenzähler benutzt. Die Schleifensteuerbefehle dekrementieren das CX-Register und veranlassen einen bedingten Sprung zu einem Zielbefehl, solange CX von 0 verschieden ist. Der Zielbefehl ist also Beginn einer Schleife, die CX-mal durchlaufen wird. Bei einigen Befehlen kann außerdem noch das Nullflag Z als Endebedingung ausgetestet werden. Ist eine Endebedingung erreicht, wird das Programm mit dem auf den Schleifenbefehl folgenden Befehl fortgesetzt.
Wie bei den bedingten Sprüngen muß der Operand eine Marke im Abstand +127 bis -128 Byte sein (Typ "short").
Eine Sonderstellung nimmt der Befehl JCXZ ein, der ebenfalls das CX-Register austestet, aber nicht dekrementiert.

LOOP target

Springe zum Schleifenbeginn

Der LOOP-Befehl erniedrigt den Inhalt des CX-Registers um den Wert 1 und veranlaßt einen Sprung zum Zielbefehl im Abstand +127 bis -128 Bytes, falls CX von 0 verschieden ist. Andernfalls wird der auf LOOP folgende Befehl abgearbeitet.

Beispiel:

```
        MOV CX,LENGTH BYTE
        MOV AX,0
        MOV SI,AX
CHECK:  ADD AX,BYTE[SI]
        INC SI
        LOOP CHECK
        MOV CHECKSUM,AX
```

Dieses Programm bildet die Prüfsumme einer Zahl von Bytes, die durch das Attribut LENGTH der Variablen BYTE bestimmt ist.

LOOPE / LOOPZ target

Springe zum Schleifenbeginn, falls "gleich" / "Null" erfüllt ist

Dieser Befehl erniedrigt den Inhalt von CX um den Wert 1 und veranlaßt einen Sprung zum Zielbefehl im Abstand +127 bis -128 Bytes, falls CX von 0 verschieden ist und das Nullflag Z auf 1 gesetzt ist.
Andernfalls wird der auf LOOPE / LOOPZ folgende Befehl abgearbeitet.

Beispiel:

```
        MOV CX,LENGTH LIST
        MOV SI,-1
FIND:   INC SI
        CMP LIST[SI],0
        LOOPZ FIND
        JNZ NONZERO
```

Dieses Programm durchsucht eine Liste nach dem ersten von 0 verschiedenen Element. Ist die ganze Liste 0, so wird mit dem auf JNZ folgenden Befehl fortgesetzt. Ansonsten wird die Schleife verlassen und zu NONZERO gesprungen. Die Adresse des gesuchten Elements ist in SI enthalten.

LOOPNE / LOOPNZ target

Springe zum Schleifenbeginn, falls "nicht gleich" / "nicht Null" erfüllt ist

Dieser Befehl erniedrigt den Inhalt des CX-Registers um den Wert 1 und veranlaßt einen Sprung zum Zielbefehl, falls CX von 0 verschieden ist und das Nullflag Z auf 0 gesetzt ist.
Andernfalls wird der auf LOOPNE / LOOPNZ folgende Befehl abgearbeitet.

JCXZ target

Springe zur Zieladresse, falls CX Null ist

Dieser Befehl testet das CX-Register wie die Schleifenbefehle, dekrementiert es aber nicht. Falls CX den Wert 0 hat, wird zur Zieladresse gesprungen. Ansonsten wird der folgende Befehl abgearbeitet.
Dieser Befehl kann am Beginn von Schleifen verwendet werden, wenn der Schleifenzähler variabel ist. Ist CX gleich zu Beginn Null, so wird die Schleife übersprungen (oder Null-mal ausgeführt).

4.6.5. Interruptbefehle

Die Interruptbefehle ermöglichen die Aktivierung von Interruptprozeduren in ähnlicher Weise wie externe (Hardware-) Interrupts. Die Reaktion des K1810WM86 auf Interrupts (Unterbrechungsanforderungen) ist ausführlich im Abschnitt 2.5 beschrieben. Sie ist für Hard- und Software-Interrupts gleich; mit der Ausnahme, daß nach Interruptbefehlen kein Anerkennungszyklus ausgeführt wird, da der Interruptvektor im Befehlskode enthalten ist. Software-Interrupts sind höher priorisiert als externe Interrupts.
Interrupts sind mit Prozeduraufrufen vom Typ "far" vergleichbar. Anstelle einer Prozedur- oder Pointervereinbarung ist jedoch lediglich das Laden der Interruptvektortabelle Voraussetzung für eine korrekte Ausführung. Der prinzipielle Ablauf ist folgender:

1. Die Abarbeitung des aktuellen Programms wird unterbrochen, die Flags und die Rückkehradresse werden in den Stapel transportiert. Das Interrupt- und das Trapflag (I und T) werden auf 0 gesetzt. CS und IP werden mit der Startadresse der Interruptprozedur geladen.
2. Die Prozedur wird abgearbeitet, bis sie durch einen Rückkehrbefehl IRET beendet wird. Dieser rekonstruiert IP, CS und die Flags aus dem Stapel.
3. Die Programmabarbeitung wird mit dem auf den Interrupt folgenden Befehl fortgesetzt.

Als Operand wird eine Typnummer (8-Bit-Wert) angegeben, die auf einen der 256 Pointer in der Interruptvektortabelle weist.

INT typ
Unterbrich

Der Befehl INT bewirkt einen unbedingten Sprung zu einem Zielbefehl, dessen Speicheradresse aus der Interruptvektortabelle entnommen wird.
Der Operand ist eine Typnummer. Die Adresse des Pointers wird durch Multiplikation der Typnummer mit 4 ermittelt. Für die Typen 3 und 4 ist die Typnummer implizit im Operationskode enthalten.
Bei der Abarbeitung eines INT-Befehls werden zunächst die Flags im gleichen Format wie beim Befehl PUSHF in den Stapel übertragen, anschließend wird das Interrupt- und Trapflag auf 0 gesetzt. Dadurch sind externe Interrupts über das Signal INTR verboten und ein eventueller Einzelschritt-Testbetrieb ausgeschaltet (vgl. Abschn. 2.5.3). Dann wird der Inhalt von CS in den Stapel übertragen und das Kodesegmentwort des Pointers geladen. Zuletzt wird der Inhalt von IP in den Stapel transportiert und das Offsetwort geladen.

Beispiele: (1) INT 192

Diese Interrupts sind gut zum Testen von Prozeduren geeignet, die für die Bedienung von externen Geräteanforderungen geschrieben

wurden. Eine andere Anwendung sind "Systemrufe", mit denen ein Anwenderprogramm die Dienste eines Betriebssystems in Anspruch nehmen kann. Für jeden Dienst stellt das System eine Prozedur bereit, die über einen Interrupt aufgerufen werden kann. Das Anwenderprogramm muß dann lediglich die Nummer des Rufs kennen und benötigt keine Adreßinformationen.

(2) INT 3

Dieser spezielle Befehl ist gut zur Programmtestung geeignet, da mit ihm Haltepunkte (Break Points) gesetzt werden können. Dazu muß der Befehl nach der Stelle, an der gestoppt werden soll, durch diesen Einbyte-Befehl ersetzt werden. Die Interruptprozedur kann dann zur Anzeige der Systemzustände genutzt werden.

INTO
Unterbrich, falls das Überlaufflag gesetzt ist

Dieser spezielle Interruptbefehl generiert einen Interrupt vom Typ 4, falls das Überlaufflag O auf 1 gesetzt ist. Es wird also ein bedingter Sprung zu einer Speicheradresse ausgeführt. Bei erfüllter Sprungbedingung ist der Ablauf identisch mit dem normalen INT-Befehl. Andernfalls wird der auf INTO folgende Befehl abgearbeitet.
Dieser Befehl kann anstelle eines bedingten Sprungs zum Testen des Überlaufflags angewendet werden, etwa zur Fehlerbehandlung.

IRET
Kehre aus der Interruptprozedur zurück

Dieser Befehl wirkt ähnlich wie der RET-Befehl am Ende einer mit CALL aufgerufenen Prozedur. Der Unterschied besteht darin, daß außer IP und CS auch die Flags aus dem Stapel rekonstruiert werden. Der Befehl bewirkt also einen unbedingten Sprung zur Rückkehradresse, wobei der Zustand der Flags wiederhergestellt wird, der zum Zeitpunkt des Auftretens des Interrupts bestand. (Insbesondere werden die Flags I und T wieder gesetzt, falls sie vor dem Sprung in die Interruptprozedur gesetzt waren.)
Mit diesem Befehl müssen alle Interruptprozeduren abgeschlossen werden, auch die für Hardware-Interrupts.

4.7. Befehle zur Prozessorsteuerung

Die Befehle dieser letzten Gruppe dienen der Steuerung verschiedener Funktionen des K1810WM86. Es werden keine Operanden beeinflußt. In dieser Gruppe sollen auch die Präfixe LOCK und Segmentoverride behandelt werden. (Das Repeat-Präfix wurde bereits im Abschn. 4.5.2 besprochen.)

4.7.1. Befehle zur Flagbeeinflussung

Mit diesen Befehlen können das Richtungsflag D, das Übertragsflag C und das Interruptflag I manipuliert werden.

CLD
Setze das Richtungsflag zurück

STD
Setze das Richtungsflag

STD und CLD werden bei Stringbefehlen angewendet. Sie bestimmen, ob die Indexregister zur Adressierung der Stringelemente aufwärts oder abwärts gezählt werden (vgl. Abschn. 4.5).

CLI
Setze das Interruptflag zurück

Wenn I auf 0 gesetzt ist, werden vom K1810WM86 keine externen Unterbrechungsanforderungen angenommen, die über das Signal /INTR signalisiert werden. (Maskierbare Interrupts werden verboten.) Alle anderen Interrupts werden nicht beeinflußt. Bei der Anerkennung eines Interrupts wird I automatisch auf 0 gesetzt. Nach RESET ist ebenfalls I = 0.

STI
Setze das Interruptflag

STI gestattet es, maskierbare Interrupts zuzulassen. Diese sind normalerweise verboten:

- nach RESET
- nach der Anerkennung eines externen oder internen Interrupts
- nach einem Befehl CLI

CLC
Setze das Übertragsflag zurück

STC
Setze das Übertragsflag

CMC
Negiere das Übertragsflag

Dieser Befehl bildet das Komplement des Übertragsflags C, das heißt, es wird auf 1 gesetzt, wenn es 0 war und umgekehrt.

4.7.2. Befehle zur externen Synchronisation

Diese Befehlsklasse umfaßt die Befehle, die den Prozessor in einen passiven Zustand versetzen oder keine Operation bewirken.

HLT
Gehe in den HALT-Zustand

Nach der Abarbeitung dieses Befehls nimmt der K1810WM86 den HALT-Zustand ein (vgl. Abschn. 2.2.2). Dieser wird erst verlassen, wenn entweder der RESET- Eingang aktiviert wird oder ein externer Interrupt auf der NMI-Leitung oder - falls das Interruptflag gesetzt ist - auf der INTR-Leitung empfangen wird.
Dieser Befehl kann benutzt werden, um Programme abzubrechen oder zu stoppen, wenn auf einen Interrupt gewartet werden muß.

WAIT
Warte, bis der TEST-Eingang aktiv ist

Der WAIT-Befehl veranlaßt den K1810WM86, den passiven Zustand einzunehmen, falls der /TEST-Eingang inaktiv ist. Nach je 5 Takten wird dieses Signal getestet. Ist es aktiv, wird das Programm mit dem nächsten Befehl fortgesetzt.
Dieser Befehl ist besonders für die Zusammenarbeit mit Coprozessoren geeignet. Dieser signalisiert am /TEST-Eingang, ob er eine übertragene Operation beendet hat oder nicht, während der K1810WM86 seine Programmbearbeitung mit einem WAIT-Befehl unterbricht, falls er das Ergebnis der Operation benötigt (vgl. auch Abschn. 2.6.4).

NOP
Führe eine Leeroperation aus

Der NOP-Befehl veranlaßt den K1810WM86, eine Leeroperation auszuführen, das heißt mit anderen Worten, nichts zu tun.
(Der Operationskode entspricht dem Befehl XCHG AX,AX.)

ESC code,src
Erweitere den Befehlssatz

Dieser Befehl ("Escape") wird vom K1810WM86 eingelesen, ohne daß eine Operation bewirkt wird. Ist ein Speicheroperand angegeben, so wird auch dieser eingelesen und wieder vergessen. Der Sinn dieses Befehls besteht darin, daß damit der Befehlssatz um 64 Grundbefehle (Operationskodes) erweitert werden kann. Diese Befehle werden von einem Coprozessor (zum Beispiel K1810WM87), der am lokalen Bus des K1810WM86 "mithört", interpretiert und ausgeführt. Zu diesem Zweck prüft er die eingelesenen Befehle auf den ESC-Kode. Ein übertragener Speicheroperand wird ebenfalls vom

Coprozessor übernommen. Auf diese Weise kann der K1810WM86 Operationen für den Coprozessor initiieren und diesen mit den notwendigen Kodes und Operandenadressen versorgen.

LOCK

Blockiere den Bus

LOCK ist kein Befehl, sondern ein Präfixbyte. Es bewirkt die Aktivierung des /LOCK-Signals für die Dauer des Befehls, der mit diesem Präfix versehen ist. (Der K1810WM86 muß im Maximum-Mode konfiguriert sein.) Alle Busanforderungen werden ignoriert, bis der mit LOCK versehene Befehl ausgeführt wurde.
Zur Anwendung dieses Befehls vergleiche auch XCHG-Befehl (Abschn. 4.2.1).

SR:

Überschreibe das Segmentregister

Dieses Präfixbyte dient der Änderung der Zuordnung der Standard-Segmentregister bei der Adressierung von Speicheroperanden (siehe auch Abschn. 3.4.2).

Mnemonik: ES:
CS:
SS:
DS:

Beispiele:
```
MOV AX,ES:DATA
MOV DL,DS:[BP]
MOVS DEST,CS:CODE
```

Nicht angewendet werden kann dieses Präfix bei Stringoperationen für den Zieloperanden.

5. Befehle des K1810WM86 im Überblick

Dieser Abschnitt gibt einen Überblick über alle Befehle des K1810WM86. Die alphabetische Befehlsliste informiert über die Operationen, Ausführungszeiten, Maschinenkodes und Flagbeeinflussungen.

5.1. Übersicht

Der Befehlssatz des K1810WM86 kann als auf zwei Ebenen existierend betrachtet werden: 300 Befehle auf Maschinenebene können zu etwa 100 Befehlen auf Assemblerniveau zusammengefaßt werden. Die große Zahl von Maschinenbefehlen resultiert aus dem Bestreben, Einfachheit mit Effektivität zu verbinden: Einfachheit im Sinne einheitlicher Mnemoniks für Befehle mit unterschiedlichen Operanden, Effektivität durch optimierte Operationskodes, um Speicherbedarf und Programmlaufzeiten zu minimieren. So können für einen bestimmten Assemblerbefehl je nach Operandentyp Maschinenbefehle von ein bis sechs Byte Länge generiert werden.
Zur Programmierung ist natürlich in erster Linie die Assemblerebene von Interesse. Die Maschinenkodes werden automatisch generiert. Sollen aber die Vorteile der Assemblerprogrammierung mit der Leistungsfähigkeit des Befehlssatzes gepaart werden, so ist es für den Programmierer von Nutzen, wenn er die besonderen Eigenschaften eines jeden Befehls kennt.

Die meisten Befehle können Byte- oder Wortdaten verarbeiten. Einige Befehle (CALL; JMP) arbeiten auch mit Doppelwortzeigern als Speichervariable. Der Assembler ermittelt den richtigen Maschinenbefehl aus dem Operandentyp. Deshalb muß allen Speichervariablen durch eine Assemblerdirektive ein Typ zugewiesen werden (vgl. auch Abschn. 6).

Die ausführlichen Befehlsbeschreibungen enthalten folgende Informationen:

1. Assemblerbefehl
2. Beschreibung der Operation
3. Flagbeeinflussung
4. Mnemonik und Operanden der Befehlstypen
5. Ausführungszeiten
6. Buszyklen
7. Maschinenkodes der Befehlstypen

Für die Benutzung der Befehlsliste sollten folgende Hinweise beachtet werden:

1. Die Mnemoniks können assemblerspezifisch von den hier verwendeten abweichen. Insbesondere unterscheiden manche Assembler Intra- und Intersegmenttransfers (CALL und CALLF, JMP und JMPF). Derartige Besonderheiten werden hier nicht berücksichtigt.

2. Die verwendeten Operandensymbole sind in Tafel 5.1 zusammengefaßt. Es werden Pseudomnemoniks verwendet, aus denen der Operandentyp auch ohne nähere Erläuterung ersichtlich wird.

Tafel 5.1. Symbole der Operanden

Symbol	Bedeutung	Anwendung
ac	Akkumulator (AX oder AL)	allgemein
code	6-Bit-Kode eines Coprozessorbefehls	ESC
count	Bitzählwert, entweder 1 oder CL	Verschieben
data	unmittelbarer Operand (Wort oder Byte)	allgemein
data8	unmittelbarer Byteoperand	allgemein
data16	unmittelbarer Wortoperand	allgemein
deststr	Name eines Zielstrings	String-Befehle
farlab	Marke außerhalb des aktuellen Kodesegments	JMP
farproc	Name einer Prozedur außerhalb des aktuellen Kodesegments	CALL
mem	Speicheroperand	allgemein
mem8	Bytevariable	allgemein
mem16	Wortvariable	allgemein
mptr16	Speicherwort, welches eine Offsetadresse enthält (Wortpointer)	CALL, JMP
mptr32	Speicherdoppelwort, welches eine Offset- und Basisadresse enthält	CALL, JMP LES, LDS
nearlab	Marke innerhalb des aktuellen Kodesegments, die mehr als -128 bis +127 Bytes entfernt ist	JMP
nearproc	Name einer Prozedur innerhalb des aktuellen Kodesegments	CALL
popvalue	Wert, der zum Inhalt von SP addiert wird	RET
port	Adresse eines Ein-/Ausgabeports	IN, OUT
port8	unmittelbare 8-Bit-Adresse eines Ein-/Ausgabeports	IN, OUT
regptr	16-Bit-Register, welches eine Offsetadresse enthält	CALL, JMP
reg	Registeroperand	allgemein
reg8	8-Bit Register (AL, BL, CL, DL, AH, BH, CH, DH)	allgemein
reg16	16-Bit-Register (AX, BX, CX, DX, DI, SI, SP, BP)	allgemein
segreg	Segmentregister (CS, DS, ES, SS)	Datentransfer
shortlab	Marke im Abstand von -128 bis +127 Bytes	Sprungbefehle, Schleifen
srcstr	Name eines Quellstrings	Stringbefehle
tab	Name einer 256-Byte-Tabelle	XLAT
typ	unmittelbarer 8-Bit-Wert, der den Interrupttyp angibt	INT

3. Die Befehle können ein bis sechs Byte lang sein. Die Bildung der Maschinenkodes ist im dritten Kapitel (Abschn. 3.6) erläutert. Tafel 5.2 enthält die in den Beschreibungen verwendeten Symbole für die Bytes 3 bis 6. Symbole in Klammern bedeuten, daß die entsprechenden Bytes vorhanden sein können oder nicht, je nach Operandentyp und Adressierungsart.

Tafel 5.2. Abkürzungen, die in den Befehlskodes verwendet werden

Symbol	Bedeutung
adr-lo	niederwertiges Byte einer direkten Adresse
adr-hi	höherwertiges Byte einer direkten Adresse
disp-lo	8-Bit-Displacement oder niederwertiges Byte eines 16-Bit-Displacements (vom Adreßmodus abhängig)
disp-hi	höherwertiges Byte eines 16-Bit-Displacements
CS-lo	niederwertiges Byte des Kodesegments
CS-hi	höherwertiges Byte des Kodesegments
IP-lo	niederwertiges Byte der Programmoffsetadresse
IP-hi	höherwertiges Byte der Programmoffsetadresse
IP-disp-lo	niederwertiges Byte des Abstands vom aktuellen Befehl
IP-disp-hi	höherwertiges Byte des Abstands vom aktuellen Befehl
IP-disp	8-Bit-Abstand vom aktuellen Befehl im Bereich -128 bis +127 Bytes
SP-disp-lo	niederwertiges Byte des Abstands vom aktuellen Stapelzeiger
SP-disp-hi	höherwertiges Byte des Abstands vom aktuellen Stapelzeiger
data-lo	unmittelbarer Byteoperand (bei w=0) oder niederwertiges Byte eines Wortoperanden (bei w=1)
data-hi	höherwertiges Byte eines Wortoperanden
port	unmittelbare 8-Bit-Adresse eines Ein-/Ausgabeports im Bereich 0 bis 255
typ	unmittelbarer 8-Bit-Wert, der einen Interrupt-Typ von 0 bis 255 repräsentiert

4. Ausführungszeiten sind als Zahl der Prozessortakte angegeben. Die Zeit ergibt sich aus dem Produkt von Taktzahl und Taktperiode (200ns bei 5MHz, 125ns bei 8MHz).
   Bei allen Speicheroperationen muß die Zeit zur Berechnung der effektiven Adresse berücksichtigt werden (in den Beschreibungen durch den Zusatz "EA" enthalten). Die Tafel 5.3 gibt die Werte an, die bei den verschiedenen Adressierungsarten addiert werden müssen. Bei allen Wortoperationen mit ungeraden Adressen müssen zu diesem Wert vier weitere Takte addiert werden.

Tafel 5.3. Zeiten zur Berechnung der effektiven Adresse

Adressierungsart		MD	R/M	EA-Takte
direkt	disp16	00	110	6
indirekt	SI	00	100	5
	DI	00	101	5
	BX	00	111	5
relativ	BX+disp8	01	111	9
	BX+disp16	10	111	9
	BP+disp8	01	110	9
	BP+disp16	10	110	9
indiziert	SI+disp8	01	100	9
	SI+disp16	10	100	9
	DI+disp8	01	101	9
	DI+disp16	10	101	9
basisindiziert	BX+SI	00	000	7
	BP+DI	00	011	7
	BX+DI	00	001	8
	BP+SI	00	010	8
	BX+SI+disp8	01	000	11
	BX+SI+disp16	10	000	11
	BP+DI+disp8	01	011	11
	BP+DI+disp16	10	011	11
	BX+DI+disp8	01	001	12
	BX+DI+disp16	10	001	12
	BP+SI+disp8	01	010	12
	BP+SI+disp16	10	010	12

Anmerkungen:
1. Addiere 4 Takte, wenn sich eine ungerade Wortadresse ergibt.
2. Befehle mit Wortoperanden interpretieren die EA als Offset des niederwertigen Bytes; das höherwertige Byte wird durch EA+1 adressiert.

Eine exakte Berechnung von Programmlaufzeiten ist nicht möglich, denn die angegebenen Zeiten gelten nur unter den Bedingungen, daß der Befehl bereits in die Warteschlange (Instruction Queue) eingelesen wurde und Operandenübertragungen zu oder von Speicherplätzen bzw. Ein-/Ausgabeports während der Befehlsausführung durchgeführt werden, ohne daß die EU warten muß. Bei allen Programmtransferbefehlen müssen Befehlslesezyklen berücksichtigt werden, da die Warteschlange gelöscht wird. (Die Arbeitsweise von EU und BIU wurde im Abschnitt 2 besprochen.)
Generell läßt es sich nicht ausschließen, daß die BIU mit Datentransfers oder dem Füllen der Warteschlange in Rückstand gerät und die EU warten muß, zumal alle Busübertragungen von der Hardware abhängig sind, die zusätzliche Wartetakte verlangen kann. Im Mittel liegen die wirklichen Programmlaufzeiten etwa 5 bis 10% über der errechneten Summe der Befehlsausführungszeiten. Die Zeit für die Abarbeitung eines konkreten Programms ist dennoch stets reproduzierbar, wenn Interrupts ausgeschlossen werden. Diese exakte Laufzeit kann durch Abarbeitung auf der konkreten Hardware gemessen werden.

5. Die Zahl der Buszyklen ist abhängig von den Operandenadressen. Sind mehrere Werte angegeben, so gilt der kleinere für alle Byteübertragungen und Wortübertragungen auf geraden Adressen. Wortübertragungen auf ungerader Adresse erfordern einen zusätzlichen Zyklus je Transfer, da in diesem Fall zwei Bytes übertragen werden. Dafür gilt der in Klammern angegebene Wert. Es werden nur Buszyklen für Operandenzugriffe angegeben; Kodelesezyklen werden nicht berücksichtigt.

6. Die Flags werden von verschiedenen Befehlen unterschiedlich beeinflußt. Dies wird durch folgende Symbole gekennzeichnet:

   X : Flag wird vom Ergebnis beeinflußt
   - : Flag bleibt unbeeinflußt
   U : Flag hat undefinierten Zustand
   R : Flag wird zurückgespeichert
   0 : Flag wird auf 0 gesetzt
   1 : Flag wird auf 1 gesetzt.

5.2. Liste der K1810WM86-Befehle

Die folgende Liste (Tafel 5.4) enthält alle Befehle des K1810WM86 in alphabetischer Reihenfolge. Während im Abschnitt 4 ausschließlich Assemblerbefehle besprochen und Anwendungshinweise gegeben wurden, soll diese Liste hauptsächlich als ergänzende Information für den Anwender dienen, der sich für die Befehlskodes, Ausführungszeiten und andere spezielle Eigenschaften der K1810WM86-Befehle interessiert. Insbesondere werden die einzelnen Befehlstypen berücksichtigt, die der Assembler generieren kann.

Tafel 5.4. Liste aller K1810WM86-Befehle

AAA **ASCII-Adjust For Addition**

Operation:
1. Falls das niederwertige Halbbyte von AL kleiner als 9 und A=0 ist, bleibt AL unverändert.
2. Falls das niederwertige Halbbyte von AL größer als 9 oder A=1 ist, wird zu AL der Wert 6 und zu AH der Wert 1 addiert. A und C werden auf 1 gesetzt.
3. Das höherwertige Halbbyte von AL wird stets auf 0H gesetzt.

Flags:

O	D	I	T	S	Z	A	P	C
U	-	-	-	U	U	X	U	X

		Takte	Buszyklen
Mnemonik:	AAA	4	-

Kode: 0011 0111

AAD **ASCII-Adjust For Division**

Operation: Der Inhalt von AH wird mit 10 multipliziert und zum Inhalt von AL addiert. Das Ergebnis wird in AL abgelegt, AH wird auf 0H gesetzt.

Flags:

O	D	I	T	S	Z	A	P	C
U	-	-	-	X	X	U	X	U

		Takte	Buszyklen
Mnemonik:	AAD	60	-

Kode: 1101 0101 | 0000 1010

AAM **ASCII-Adjust For Multiplication**

Operation: Der Inhalt von AL wird durch 10 dividiert und das Ergebnis in AH abgespeichert. Der entstandene Rest (AL modulo 10) wird in AL abgespeichert.

Flags:

O	D	I	T	S	Z	A	P	C
U	-	-	-	X	X	U	X	U

		Takte	Buszyklen
Mnemonik:	AAM	83	-

Kode: 1101 0100 | 0000 1010

AAS **ASCII-Adjust For Subtraction**

Operation:
1. Falls das niederwertige Halbbyte von AL kleiner als 9 und A=0 ist, bleibt AL unverändert.
2. Falls das niederwertige Halbbyte von AL größer als 9 oder A=1 ist, wird von AL der Wert 6 und von AH der Wert 1 subtrahiert. A und C werden auf 1 gesetzt.
3. Das höherwertige Halbbyte von AL wird stets auf 0H gesetzt.

Flags:

O	D	I	T	S	Z	A	P	C
U	-	-	-	U	U	X	U	X

		Takte	Buszyklen
Mnemonik:	AAS	4	-

Kode: 0011 1111

ADC dest,src **Add With Carry**

Operation: Der Quelloperand wird zum Zieloperanden addiert.

Zur Summe wird 1 addiert, falls C=1 ist. Der Zieloperand wird durch das Ergebnis ersetzt.

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

TYP 1:

		Takte	Buszyklen
Mnemonik:	ADC reg,reg	3	-
	ADC reg,mem	9 + EA	1 (2)
	ADC mem,reg	16 + EA	2 (4)

Kode:

0001 00dw	MD REG R/M	(disp-lo)	(disp-hi)

TYP 2:

		Takte	Buszyklen
Mnemonik:	ADC ac,data	4	-

Kode:

0001 010w	data-lo	(data-hi)

TYP 3:

		Takte	Buszyklen
Mnemonik:	ADC reg,data	4	-
	ADC mem,data	17 + EA	2 (4)

Kode:

1000 00sw	MD 010 R/M	(disp-lo)	(disp-hi)
		data-lo	(data-hi)

ADD dest,src **Add**

Operation: Der Quelloperand wird zum Zieloperanden addiert. Der Zieloperand wird durch das Ergebnis ersetzt.

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

TYP 1:

Mnemonik:	Takte	Buszyklen
ADD reg,reg	3	-
ADD reg,mem	9 + EA	1 (2)
ADD mem,reg	16 + EA	2 (4)

Kode:

0000 00dw	MD REG R/M	(disp-lo)	(disp-hi)

TYP 2:

Mnemonik:	Takte	Buszyklen
ADD ac,data	4	-

Kode:

0000 010w	data-lo	(data-hi)

TYP 3:

Mnemonik:	Takte	Buszyklen
ADD reg,data	4	-
ADD mem,data	17 + EA	2 (4)

Kode:

1000 00sw	MD 000 R/M	(disp-lo)	(disp-hi)
		data-lo	(data-hi)

AND dest,src **Logical AND**

Operation: Der Quelloperand wird mit dem Zieloperanden logisch "UND"-verknüpft. Der Zieloperand wird durch das Ergebnis ersetzt.

Flags:

O	D	I	T	S	Z	A	P	C
0	-	-	-	X	X	U	X	0

TYP 1:

Mnemonik:	Takte	Buszyklen
AND reg,reg	3	-
AND reg,mem	9 + EA	1 (2)
AND mem,reg	16 + EA	2 (4)

Kode:

0010 00dw	MD REG R/M	(disp-lo)	(disp-hi)

TYP 2:

Mnemonik:	Takte	Buszyklen
AND ac,data	4	-

Kode:

0010 010w	data-lo	(data-hi)

TYP 3:		Takte	Buszyklen
Mnemonik:	AND reg,data	4	-
	AND mem,data	17 + EA	2 (4)

Kode:	1000 000w	MD 100 R/M	(disp-lo)	(disp-hi)
			data-lo	(data-hi)

CALL near-procedure **Call (Intrasegment)**

Operation:
1. IP wird in den Stapel übertragen.
2. IP wird mit der Zieladresse geladen.
3. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

TYP 1:		Takte	Buszyklen
Mnemonik:	DIREKT		
	CALL nearproc	19	1

Kode:	1110 1000	IP-disp-lo IP-disp-hi

TYP 2:		Takte	Buszyklen
Mnemonik:	INDIREKT		
	CALL regptr	16	1
	CALL mptr16	21 + EA	2 (3)

Kode:	1111 1111	MD 010 R/M	(disp-lo)	(disp-hi)

CALL far-procedure **Call (Intersegment)**

Operation:
1. CS wird in den Stapel übertragen.
2. CS wird mit der Zielsegmentadresse geladen.
3. IP wird in den Stapel übertragen.
4. IP wird mit der Zieladresse geladen.
5. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

TYP 1:		Takte	Buszyklen
Mnemonik:	DIREKT		
	CALL farproc	28	2

Kode:	1001 1010	IP-lo IP-hi CS-lo	CS-hi

TYP 2:		Takte	Buszyklen
Mnemonik:	INDIREKT		
	CALL mptr32	37 + EA	4 (6)

Kode:	1111 1111	MD 011 R/M	(disp-lo)	(disp-hi)

CBW — **Convert Byte To Word**

Operation: Falls der Inhalt von AL kleiner als 80H ist, wird AH auf 0 gesetzt, andernfalls auf FFH.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	CBW	2	-

Kode: 1001 1000

CLC — **Clear Carry Flag**

Operation: Das Übertragsflag C wird auf 0 gesetzt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	0

		Takte	Buszyklen
Mnemonik:	CLC	2	-

Kode: 1111 1000

CLD — **Clear Direction Flag**

Operation: Das Richtungsflag D wird auf 0 gesetzt.

Flags:

O	D	I	T	S	Z	A	P	C
-	0	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	CLD	2	-

Kode: 1111 1100

CLI — **Clear Interrupt Flag**

Operation: Das Interruptflag I wird auf 0 gesetzt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	0	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	CLI	2	-

Kode: 1111 1010

CMC **Complement Carry Flag**

Operation: Das Übertragsflag C wird invertiert.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	X

		Takte	Buszyklen
Mnemonik:	CMC	2	-

Kode: 1111 0101

CMP dest,src **Compare**

Operation: Der Quelloperand wird vom Zieloperanden subtrahiert. Die Operanden bleiben erhalten, die Flags werden aktualisiert.

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

TYP 1:

		Takte	Buszyklen
Mnemonik:	CMP reg,reg	3	-
	CMP reg,mem	9 + EA	1 (2)
	CMP mem,reg	9 + EA	1 (2)

Kode:	0011 10dw	MD REG R/M	(disp-lo)	(disp-hi)

TYP 2:

		Takte	Buszyklen
Mnemonik:	CMP ac,data	4	-

Kode:	0011 110w	data-lo	(data-hi)

TYP 3:

		Takte	Buszyklen
Mnemonik:	CMP reg,data	4	-
	CMP mem,data	10 + EA	1 (2)

Kode:	1000 00sw	MD 111 R/M	(disp-lo)	(disp-hi)
			data-lo	(data-hi)

CMPS deststr,srcstr **Compare String**
CMPSB / CMPSW

Operation:
1. Das Zielelement wird vom Quellelement subtrahiert. Die Operanden bleiben erhalten, die Flags werden aktualisiert.
2. SI und DI werden um 1 (w=0) oder 2 (w=1) erhöht (D=0) oder erniedrigt (D=1)

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

		Takte	Buszyklen
Mnemonik:	CMPS deststr,srcstr	22	2 (4)
	CMPSB / CMPSW		
	REP CMPS..	9 + 22/rep	2 (4)/rep
Kode:	1010 011w		

CWD **Convert Word To Double Word**

Operation: Falls der Inhalt von AX kleiner als 8000H ist, wird DX auf 0 gesetzt, andernfalls auf FFFFH.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	CWD	5	-

Kode: 1001 1001

DAA **Decimal-Adjust For Addition**

Operation:
1. Falls beide Halbbytes von AL kleiner als 9 und A=0 ist, bleibt AL unverändert.
2. Falls das niederwertige Halbbyte von AL größer als 9 oder A=1 ist, wird zu AL der Wert 6 addiert. A wird auf 1 gesetzt.
3. Falls AL größer als 9FH oder C=1 ist, wird zu AL der Wert 60H addiert und C auf 1 gesetzt.

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

		Takte	Buszyklen
Mnemonik:	DAA	4	-

Kode: 0010 0111

DAS **Decimal-Adjust For Subtraction**

Operation:
1. Falls beide Halbbytes von AL kleiner als 9 und A=0 ist, bleibt AL unverändert.
2. Falls das niederwertige Halbbyte von AL größer als 9 oder A=1 ist, wird von AL der Wert 6 subtrahiert. A wird auf 1 gesetzt.
3. Falls AL größer als 9FH oder C=1 ist, wird von AL der Wert 60H subtrahiert und C auf 1 gesetzt.

Flags:

O	D	I	T	S	Z	A	P	C
U	-	-	-	X	X	X	X	X

		Takte	Buszyklen
Mnemonik:	DAS	4	-

Kode: 0010 1111

DEC dest **Decrement**

Operation: Vom Zieloperanden wird 1 subtrahiert. Der Zieloperand wird durch das Ergebnis ersetzt.

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	-

TYP 1:

		Takte	Buszyklen
Mnemonik:	DEC reg8	3	-
	DEC mem	15 + EA	2 (4)

Kode: 1111 111w | MD 001 R/M | (disp-lo) | (disp-hi)

TYP 2:

		Takte	Buszyklen
Mnemonik:	DEC reg16	2	-

Kode: 0100 1REG

DIV src **Divide**

Operation:
1. Der Inhalt des Akkumulators und seiner Erweiterung (bei Byteoperationen AL und AH, bei Wortoperationen AX und DX) wird durch den Quelloperand dividiert. Der Quotient wird im Akkumulator (AL bzw. AX) abgelegt, der verbleibende Rest im Erweiterungsregister (AH bzw. DX).
2. Überschreitet der Quotient die Kapazität des Akkumulators, wird ein Interrupt vom Typ 0 ausgelöst. Der Akkumulator und seine Erweiterung nehmen einen undefinierten Wert an.

Flags:

O	D	I	T	S	Z	A	P	C
U	-	-	-	U	U	U	U	U

		Takte	Buszyklen
Mnemonik:	DIV reg8	80- 90	-
	DIV reg16	144-162	-
	DIV mem8	86- 96 + EA	1
	DIV mem16	150-168 + EA	1 (2)

Kode: 1111 011w | MD 110 R/M | (disp-lo) | (disp-hi)

ESC code,src **Escape**

Operation: keine

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Mnemonik:

	Takte	Buszyklen
ESC data,reg	2	-
ESC data,mem	8 + EA	1 (2)

Kode:

1101 1xxx	MD yyy R/M	(disp-lo)	(disp-hi)

xxxyyy : 6-Bit-Code für Coprozessor

HLT **Halt**

Operation: HALT

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Mnemonik:

	Takte	Buszyklen
HLT	2	-

Kode:

1111 0100

IDIV src **Integer Divide**

Operation:
1. Der Inhalt des Akkumulators und seiner Erweiterung (bei Byteoperationen AL und AH, bei Wortoperationen AX und DX) wird durch den Quelloperand unter Beachtung des Vorzeichens dividiert. Der Quotient wird im Akkumulator (AL bzw.AX) abgelegt, der verbleibende Rest im Erweiterungsregister (AH bzw. DX).
2. Überschreitet der Quotient die Kapazität des Akkumulators, so wird ein Interrupt vom Typ 0 ausgelöst. Der Akkumulator und seine Erweiterung nehmen einen undefinierten Wert an.

Flags:

O	D	I	T	S	Z	A	P	C
U	-	-	-	U	U	U	U	U

Mnemonik:

	Takte	Buszyklen
IDIV reg8	101-112	-
IDIV reg16	165-184	-
IDIV mem8	107-118 + EA	1
IDIV mem16	171-190 + EA	1 (2)

Kode:

1111 011w	MD 111 R/M	(disp-lo)	(disp-hi)

IMUL src — **Integer Multiply**

Operation: Der Quelloperand wird mit dem Akkumulatorinhalt unter Beachtung des Vorzeichens multipliziert. Das Ergebnis von doppelter Länge wird im Akkumulator und seiner Erweiterung abgespeichert (bei Byteoperationen AL und AH, bei Wortoperationen AX und DX).

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	U	U	U	U	X

Mnemonik:

	Takte	Buszyklen
IMUL reg8	80- 98	-
IMUL reg16	128-158	-
IMUL mem8	86-104 + EA	1
IMUL mem16	134-160 + EA	1 (2)

Kode:

1111 011w	MD 101 R/M	(disp-lo)	(disp-hi)

IN dest,src — **Input**

Operation: Der Inhalt eines Ports wird zum Akkumulator übertragen.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

TYP 1:
Mnemonik:

	Takte	Buszyklen
IN ac,port8	10	1 (2)

Kode:

1110 010w	port

TYP 2:
Mnemonik:

	Takte	Buszyklen
IN ac,DX	8	1 (2)

Kode:

1110 110w

INC dest — **Increment**

Operation: Zum Zieloperand wird 1 addiert. Der Zieloperand wird durch das Ergebnis ersetzt.

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	-

TYP 1:
Mnemonik:

	Takte	Buszyklen
INC reg8	3	-
INC mem	15 + EA	2 (4)

Kode:

1111 111w	MD 000 R/M	(disp-lo)	(disp-hi)

TYP 2:		Takte	Buszyklen
Mnemonik:	INC reg16	2	-
Kode:	0100 OREG		

INT typ — **Interrupt**

Operation:
1. PSW wird in den Stapel übertragen.
2. CS wird in den Stapel übertragen.
3. CS wird mit dem höherwertigen Wort des Interruptvektors geladen.
4. IP wird in den Stapel übertragen.
5. IP wird mit dem niederwertigen Wort des Interruptvektors geladen.
6. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	0	0	-	-	-	-	-

TYP 1:		Takte	Buszyklen
Mnemonik:	INT typ	51	5
Kode:	1100 1101 | typ		

TYP 2:		Takte	Buszyklen
Mnemonik:	INT 3	52	5
Kode:	1100 1100		

INTO — **Interrupt On Overflow**

Operation: Falls O = 1 ist, wird die gleiche Operation wie bei INT ausgeführt.
Falls O = 0 ist, wird keine Operation ausgeführt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	0	0	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	INTO	53 / 4	5 / -
Kode:	1100 1110		

IRET **Interrupt Return**

Operation:
1. IP wird aus dem Stapel geladen.
2. CS wird aus dem Stapel geladen.
3. PSW wird aus dem Stapel geladen.
4. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
R	R	R	R	R	R	R	R	R

		Takte	Buszyklen
Mnemonik:	IRET	24	3

Kode: 1100 1111

JA shortlab **Jump If Above**
JNBE shortlab **Jump If Not Below Or Equal**

Operation:
1. Falls C=0 und Z=0 ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	JA shortlab	16 / 4	-
	JNBE shortlab	16 / 4	-

Kode: 0111 0111 | IP-disp

JAE shortlab **Jump If Above Or Equal**
JNB shortlab **Jump If Not Below**
JNC shortlab **Jump If Not Carry**

Operation:
1. Falls C=0 ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	JAE shortlab	16 / 4	-
	JNB shortlab	16 / 4	-
	JNC shortlab	16 / 4	-

Kode: 0111 0011 | IP-disp

JB shortlab — **Jump If Below**
JNAE shortlab — **Jump If Not Above Or Equal**
JC shortlab — **Jump If Carry**

Operation:
1. Falls C=1 ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Mnemonik:

	Takte	Buszyklen
JB shortlab	16 / 4	-
JNAE shortlab	16 / 4	-
JC shortlab	16 / 4	-

Kode:

0111 0010	IP-disp

JBE shortlab — **Jump If Below Or Equal**
JNA shortlab — **Jump If Not Above**

Operation:
1. Falls C=1 oder Z=1 ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Mnemonik:

	Takte	Buszyklen
JBE shortlab	16 / 4	-
JNA shortlab	16 / 4	-

Kode:

0111 0110	IP-disp

JCXZ shortlab — **Jump If CX Zero**

Operation:
1. Falls der Inhalt von CX=0 ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Mnemonik:

	Takte	Buszyklen
JCXZ shortlab	18 / 6	-

Kode:

1110 0011	IP-disp

JE shortlab — **Jump If Equal**
JZ shortlab — **Jump If Zero**

Operation:
1. Falls Z=1 ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	JE shortlab	16 / 4	-
	JZ shortlab	16 / 4	-

Kode: 0111 0100 | IP-disp

JG shortlab — **Jump If Greater**
JNLE shortlab — **Jump If Not Less Or Equal**

Operation:
1. Falls O=S und Z=0 ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	JG shortlab	16 / 4	-
	JNLE shortlab	16 / 4	-

Kode: 0111 1111 | IP-disp

JGE shortlab — **Jump If Greater Or Equal**
JNL shortlab — **Jump If Not Less**

Operation:
1. Falls O=S ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	JGE shortlab	16 / 4	-
	JNL shortlab	16 / 4	-

Kode: 0111 1101 | IP-disp

JL shortlab — **Jump If Less**
JNGE shortlab — **Jump If Not Greater Or Equal**

Operation:
1. Falls O≠S ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Mnemonik:

	Takte	Buszyklen
JL shortlab	16 / 4	-
JNGE shortlab	16 / 4	-

Kode:

0111 1100	IP-disp

JLE shortlab — **Jump If Less Or Equal**
JNG shortlab — **Jump If Not Greater**

Operation:
1. Falls O≠S oder Z=1 ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Mnemonik:

	Takte	Buszyklen
JLE shortlab	16 / 4	-
JNG shortlab	16 / 4	-

Kode:

0111 1110	IP-disp

JMP near-target — **Jump (Intrasegment)**

Operation:
1. IP wird mit der Zieladresse geladen.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

TYP 1:
Mnemonik:

DIREKT	Takte	Buszyklen
JMP shortlab	15	-
JMP nearlab	15	-

Kode:

1110 10t1	IP-disp-lo	(IP-disp-hi)

t = 0 Typ NEAR (16-Bit-Displacement)
t = 1 Typ SHORT (8-Bit-Displacement)

TYP 2:	INDIREKT	Takte	Buszyklen
Mnemonik:	JMP regptr	11	-
	JMP mptr16	18 + EA	1 (2)

Kode:	1111 1111	MD 100 R/M	(disp-lo)	(disp-hi)

JMP far-target **Jump (Intersegment)**

Operation:
1. CS wird mit der Zielsegmentadresse geladen.
2. IP wird mit der Zieladresse geladen.
3. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

TYP 1:	DIREKT	Takte	Buszyklen
Mnemonik:	JMP farlab	15	-

Kode:	1110 1010	IP-lo	IP-hi	CS-lo	CS-hi

TYP 2:	INDIREKT	Takte	Buszyklen
Mnemonik:	JMP mptr32	24 + EA	2 (4)

Kode:	1111 1111	MD 101 R/M	(disp-lo)	(disp-hi)

JNE shortlab **Jump If Not Equal**
JNZ shortlab **Jump If Not Zero**

Operation:
1. Falls Z=0 ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	JNE shortlab	16 / 4	-
	JNZ shortlab	16 / 4	-

Kode:	0111 0101	IP-disp

JNO shortlab **Jump If Not Overflow**

Operation:
1. Falls O=0 ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	JNO shortlab	16 / 4	-

Kode: 0111 0001 | IP-disp

JNP shortlab **Jump If Not Parity**
JPO shortlab **Jump If Parity Odd**

Operation:
1. Falls P=0 ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	JNP shortlab	16 / 4	-
	JPO shortlab	16 / 4	-

Kode: 0111 1011 | IP-disp

JNS shortlab **Jump If Not Sign**

Operation:
1. Falls S=0 ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	JNS shortlab	16 / 4	-

Kode: 0111 1001 | IP-disp

JO shortlab **Jump If Overflow**

Operation:
1. Falls O=1 ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	JO shortlab	16 / 4	-

Kode: 0111 0000 | IP-disp

JP shortlab **Jump If Parity**
JPE shortlab **Jump If Parity Even**

Operation: 1. Falls P=1 ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	JP shortlab	16 / 4	-
	JPE shortlab	16 / 4	-

Kode: 0111 1010 | IP-disp

JS shortlab **Jump If Sign**

Operation: 1. Falls S=1 ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	JS shortlab	16 / 4	-

Kode: 0111 1000 | IP-disp

LAHF **Load AH From Flags**

Operation: Das niederwertige Byte des PSW wird nach AH übertragen.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	LAHF	4	-

Kode: 1001 1111

LDS dest,src — **Load Pointer Using DS**

Operation: Das niederwertige Wort des Quelloperanden (32-Bit-Pointer) wird zum Zieloperanden (16-Bit-Register) übertragen. Das höherwertige Wort des Quelloperanden wird nach DS übertragen.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	LDS reg16,mptr32	16 + EA	2 (4)

Kode:

1100 0101	MD REG R/M	(disp-lo)	(disp-hi)

MD = 11 verboten

LEA dest,src — **Load Effective Address**

Operation: Der Zieloperand (16-Bit-Register) wird mit der effektiven Adresse des Quelloperanden (Speicherzelle) geladen.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	LEA reg16,mem	2 + EA	-

Kode:

1000 1101	MD REG R/M	(disp-lo)	(disp-hi)

MD = 11 verboten

LES dest,src — **Load Pointer Using ES**

Operation: Das niederwertige Wort des Quelloperanden (32-Bit-Pointer) wird zum Zieloperanden (16-Bit-Register) übertragen. Das höherwertige Wort des Quelloperanden wird nach ES übertragen.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	LES reg16,mptr32	16 + EA	2 (4)

Kode:

1100 0100	MD REG R/M	(disp-lo)	(disp-hi)

MD = 11 verboten

LOCK **(Prefix Byte)** **Lock**

Operation: Im folgenden Befehl wird der LOCK-Ausgang aktiv.

		Takte	Buszyklen
Mnemonik:	LOCK	2	-

Kode: 1111 0000

LODS srcstr **Load String**
LODSB / LODSW

Operation:
1. Das Quellelement wird zum Akkumulator übertragen.
2. SI und DI werden um 1 (w=0) oder 2 (w=1) erhöht (D=0) oder erniedrigt (D=1).

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	LODS srcstr	12	1 (2)
	LODSB / LODSW		
	REP LODS..	9 + 13/rep	1 (2)/rep

Kode: 1010 110w

LOOP shortlab **Loop**

Operation:
1. CX wird um 1 erniedrigt.
2. Falls CX≠0 ist, wird zu IP das Displacement addiert.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	LOOP shortlab	17 / 5	-

Kode: 1110 0010 | IP-disp

LOOPE shortlab **Loop If Equal**
LOOPZ shortlab **Loop If Zero**

Operation:
1. CX wird um 1 erniedrigt
2. Falls Z=1 und CX≠0 ist, wird zu IP das Displacement addiert.
3. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	LOOPE shortlab	18 / 6	-
	LOOPZ shortlab	18 / 6	-

Kode: 1110 0001 | IP-disp

LOOPNE shortlab **Loop If Not Equal**
LOOPNZ shortlab **Loop If Not Zero**

Operation:
1. CX wird um 1 erniedrigt.
2. Falls Z=0 und CX≠0 ist, wird zu IP das Displacement addiert.
3. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	LOOPNE shortlab	19 / 5	-
	LOOPNZ shortlab	19 / 5	-

Kode: 0111 0100 | IP-disp

MOV dest,src **Move**

Operation: Der Quelloperand wird zum Zieloperand übertragen.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

TYP 1:

		Takte	Buszyklen
Mnemonik:	MOV reg,reg	2	-
	MOV reg,mem	8 + EA	1 (2)
	MOV mem,reg	9 + EA	1 (2)

Kode: 1000 10dw | MD REG R/M | (disp-lo) | (disp-hi)

TYP 2:

		Takte	Buszyklen
Mnemonik:	MOV ac,mem	10	1 (2)
	MOV mem,ac	10	1 (2)

Kode: 1010 00dw | adr-lo | adr-hi

TYP 3:

		Takte	Buszyklen
Mnemonik:	MOV segreg,reg16	2	-
	MOV segreg,mem16	8 + EA	1 (2)
	MOV reg16,segreg	2	-
	MOV mem16,segreg	9 + EA	1 (2)

Kode: 1000 11d0 | MD 0SR R/M | (disp-lo) | (disp-hi)
SR = 01 bei d = 1 nicht erlaubt

TYP 4:		Takte	Buszyklen
Mnemonik:	MOV reg,data	4	-

Kode:	1011 wREG	data-lo	(data-hi)

TYP 5:		Takte	Buszyklen
Mnemonik:	MOV mem,data	10 +EA	1 (2)

Kode:	1100 011w	MD 000 R/M	(disp-lo)	(disp-hi)
			data-lo	(data-hi)

MOVS deststr,srcstr **Move String**
MOVSB / MOVSW

Operation: 1. Das Quellelement wird zum Zielelement übertragen.
2. SI und DI werden um 1 (w=0) oder 2 (w=1) erhöht (D=0) oder erniedrigt (D=1).

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	MOVS deststr,srcstr	18	2 (4)
	MOVSB / MOVSW		
	REP MOVS..	9 + 17/rep	2 (4)/rep

Kode:	1010 010w

MUL src **Multiply**

Operation: Der Quelloperand wird mit dem Akkumulatorinhalt multipliziert. Das Ergebnis von doppelter Länge wird im Akkumulator und seiner Erweiterung abgespeichert (bei Byteoperationen AL und AH, bei Wortoperationen AX und DX).

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	U	U	U	U	X

		Takte	Buszyklen
Mnemonik:	MUL reg8	70- 77	-
	MUL reg16	118-133	-
	MUL mem8	76-83 + EA	1
	MUL mem16	124-139 + EA	1 (2)

Kode:	1111 011w	MD 100 R/M	(disp-lo)	(disp-hi)

NEG dest — **Negate**

Operation: Der Zieloperand wird von 0 subtrahiert und durch das Ergebnis ersetzt.

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	1

(falls dest=0 wird C=0 gesetzt)

Mnemonik:

	Takte	Buszyklen
NEG reg	3	-
NEG mem	16 + EA	2 (4)

Kode:

1111 011w	MD 011 R/M	(disp-lo)	(disp-hi)

NOP — **No Operation**

Operation: keine

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Mnemonik:

	Takte	Buszyklen
NOP	3	-

Kode:

1001 0000

NOT dest — **Logical NOT**

Operation: Der Zieloperand wird negiert (Einerkomplement).

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Mnemonik:

	Takte	Buszyklen
NOT reg	3	-
NOT mem	16 + EA	2 (4)

Kode:

1111 011w	MD 010 R/M	(disp-lo)	(disp-hi)

OR dest,src — **Logical OR**

Operation: Der Quelloperand wird mit dem Zieloperanden logisch "ODER"-verknüpft. Der Zieloperand wird durch das Ergebnis ersetzt.

Flags:

O	D	I	T	S	Z	A	P	C
0	-	-	-	X	X	U	X	0

TYP 1:		Takte	Buszyklen
Mnemonik:	OR reg,reg	3	-
	OR reg,mem	9 + EA	1 (2)
	OR mem,reg	16 + EA	2 (4)

Kode:	0000 10dw	MD REG R/M	(disp-lo)	(disp-hi)

TYP 2:		Takte	Buszyklen
Mnemonik:	OR ac,data	4	-

Kode:	0000 110w	data-lo	(data-hi)

TYP 3:		Takte	Buszyklen
Mnemonik:	OR reg,data	4	-
	OR mem,data	17 + EA	2 (4)

Kode:	1000 000w	MD 001 R/M	(disp-lo)	(disp-hi)
			data-lo	(data-hi)

OUT dest,src **Output**

Operation: Der Akkumulatorinhalt wird zu einem Port übertragen.

Flags:	O	D	I	T	S	Z	A	P	C
	-	-	-	-	-	-	-	-	-

TYP 1:		Takte	Buszyklen
Mnemonik:	OUT port8,ac	10	1 (2)

Kode:	1110 011w	port

TYP 2:		Takte	Buszyklen
Mnemonik:	OUT DX,ac	8	1 (2)

Kode:	1110 111w

POP dest **Pop**

Operation: Der Inhalt der Stapelspitze wird zum Zieloperanden übertragen.

Flags:	O	D	I	T	S	Z	A	P	C
	-	-	-	-	-	-	-	-	-

TYP 1:		Takte	Buszyklen
Mnemonik:	POP reg16	8	1

Kode:	0101 1REG

TYP 2:		Takte	Buszyklen
Mnemonik:	POP segreg	8	1

Kode: 000 SR 111
SR = 01 verboten

TYP 3:		Takte	Buszyklen
Mnemonik:	POP mem16	17 + EA	2 (3)

Kode:	1000 1111	MD 000 R/M	(disp-lo)	(disp-hi)

POPF **Pop Flags**

Operation: Der Inhalt der Stapelspitze wird in das PSW übertragen.

Flags:

O	D	I	T	S	Z	A	P	C
R	R	R	R	R	R	R	R	R

		Takte	Buszyklen
Mnemonik:	POPF	8	1

Kode: 1001 1101

PUSH src **Push**

Operation: Der Quelloperand wird zur Stapelspitze übertragen.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

TYP 1:		Takte	Buszyklen
Mnemonik:	PUSH reg16	11	1

Kode: 0101 OREG

TYP 2:		Takte	Buszyklen
Mnemonik:	PUSH segreg	10	1

Kode: 000 SR 110

TYP 3:		Takte	Buszyklen
Mnemonik:	PUSH mem16	16 + EA	2 (3)

Kode:	1111 1111	MD 110 R/M	(disp-lo)	(disp-hi)

PUSHF **Push Flags**

Operation: Der Inhalt des PSW wird zur Stapelspitze übertragen.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	PUSHF	10	1

Kode: 1001 1100

RCL dest,count **Rotate Left Through Carry**

Operation:
1. Alle Bits werden um eine Stelle nach links verschoben. C wird in die niederwertigste Bitposition geschoben, das höchstwertige Bit in C.
2. Der Wert von count wird um 1 erniedrigt.
3. Die Operation wird wiederholt, solange count von 0 verschieden ist.

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	-	-	-	-	X

		Takte	Buszyklen
Mnemonik:	RCL reg,1	2	-
	RCL mem,1	15 + EA	2 (4)
	RCL reg,CL	8 + 4/count	-
	RCL mem,CL	20 + EA + 4/count	2 (4)

Kode:

1101 00cw	MD 010 R/M	(disp-lo)	(disp-hi)

c = 0 count = 1
c = 1 count = CL

RCR dest,count **Rotate Right Through Carry**

Operation:
1. Alle Bits werden um eine Stelle nach rechts verschoben. C wird in die höchstwertige Bitposition geschoben, das niederwertigste Bit in C.
2. Der Wert von count wird um 1 erniedrigt.
3. Die Operation wird wiederholt, solange count von 0 verschieden ist.

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	-	-	-	-	X

Mnemonik:

	Takte	Buszyklen
RCR reg,1	2	-
RCR mem,1	15 + EA	2 (4)
RCR reg,CL	8 + 4/count	-
RCR mem,CL	20 + EA + 4/count	2 (4)

Kode:

1101 00cw	MD 011 R/M	(disp-lo)	(disp-hi)

c = 0 count = 1
c = 1 count = CL

REP (Prefix Byte) **Repeat**
REPE (Prefix Byte) **Repeat While Equal**
REPZ (Prefix Byte) **Repeat While Zero**

Operation:
1. CX wird um 1 erniedrigt.
2. Falls CX≠0 und bei CMPS oder SCAS Z = 1 ist, wird die Stringoperation wiederholt.

Mnemonik:

	Takte	Buszyklen
REP	2	-
REPE	2	-
REPZ	2	-

Kode:

1111 0011

REPNE (Prefix Byte) **Repeat While Not Equal**
REPNZ (Prefix Byte) **Repeat While Not Zero**

Operation:
1. CX wird um 1 erniedrigt.
2. Falls Z=0 und CX≠0 ist, wird die Stringoperation wiederholt.

Mnemonik:

	Takte	Buszyklen
REPNE	2	-
REPNZ	2	-

Kode:

1111 0010

RET **Return (Intrasegment)**

Operation:
1. IP wird aus dem Stapel geladen.
2. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Mnemonik:

	Takte	Buszyklen
RET	8	1

Kode:

1100 0011

RET **Return (Intersegment)**

Operation:
1. IP wird aus dem Stapel geladen.
2. CS wird aus dem Stapel geladen.
3. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	RET	18	2

Kode: 1100 1011

RET popvalue **Return And Pop (Intrasegment)**

Operation:
1. IP wird aus dem Stapel geladen.
2. Zu SP wird das Displacement addiert.
3. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	RET popvalue	12	1

Kode: 1100 0010 | SP-disp-lo | SP-disp-hi

RET popvalue **Return And Pop (Intersegment)**

Operation:
1. IP wird aus dem Stapel geladen.
2. CS wird aus dem Stapel geladen.
3. Zu SP wird das Displacement addiert.
4. Der durch CS:IP adressierte Befehl wird geholt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	RET popvalue	17	2

Kode: 1100 1010 | SP-disp-lo | SP-disp-hi

ROL dest,count **Rotate Left**

Operation:
1. Alle Bits werden um eine Stelle nach links verschoben. Das höchstwertige Bit wird in C und an die niederwertigste Bitposition geschoben.
2. Der Wert von count wird um 1 erniedrigt.
3. Die Operation wird wiederholt, solange count von 0 verschieden ist.

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	-	-	-	-	X

Mnemonik:

	Takte	Buszyklen
ROL reg,1	2	-
ROL mem,1	15 + EA	2 (4)
ROL reg,CL	8 + 4/count	-
ROL mem,CL	20 + EA + 4/count	2 (4)

Kode:

1101 00cw	MD 000 R/M	(disp-lo)	(disp-hi)

c = 0 count = 1
c = 1 count = CL

ROR dest,count **Rotate Right**

Operation:
1. Alle Bits werden um eine Stelle nach rechts verschoben. Das niederwertigste Bit wird in C und an die höchstwertige Bitposition geschoben.
2. Der Wert von count wird um 1 erniedrigt.
3. Die Operation wird wiederholt, solange count von 0 verschieden ist.

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	-	-	-	-	X

Mnemonik:

	Takte	Buszyklen
ROR reg,1	2	-
ROR mem,1	15 + EA	2 (4)
ROR reg,CL	8 + 4/count	-
ROR mem,CL	20 + EA + 4/count	2 (4)

Kode:

1101 00cw	MD 001 R/M	(disp-lo)	(disp-hi)

c = 0 count = 1
c = 1 count = CL

SAHF **Store AH Into Flags**

Operation: Das niederwertige Byte des PSW wird mit dem Inhalt von AH geladen.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	R	R	R	R	R

Mnemonik:

	Takte	Buszyklen
SAHF	4	

Kode:

1001 1110

SAL dest,count **Shift Arithmetic Left**
SHL dest,count **Shift Logical Left**

Operation:
1. Alle Bits werden um eine Stelle nach links verschoben. Das höchstwertige Bit wird in C geschoben. Die niederwertigste Bitposition wird auf 0 gesetzt.
2. Der Wert von count wird um 1 erniedrigt.
3. Die Operation wird wiederholt, solange count von 0 verschieden ist.

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	U	X	X

Mnemonik:

	Takte	Buszyklen
SAL reg,1	2	-
SAL mem,1	15 + EA	2 (4)
SAL reg,CL	8 + 4/count	-
SAL mem,CL	20 + EA + 4/count	2 (4)
SHL reg,1	2	-
SHL mem,1	15 + EA	2 (4)
SHL reg,CL	8 + 4/count	-
SHL mem,CL	20 + EA + 4/count	2 (4)

Kode:

1101 00cw	MD 100 R/M	(disp-lo)	(disp-hi)

c = 0 count = 1
c = 1 count = CL

SAR dest,count **Shift Arithmetic Right**

Operation:
1. Alle Bits werden um eine Stelle nach rechts verschoben. Das niederwertigste Bit wird in C geschoben. Die höchstwertige Bitposition behält den Wert, den sie vor dem Verschieben hatte.
2. Der Wert von count wird um 1 erniedrigt.
3. Die Operation wird wiederholt, solange count von 0 verschieden ist.

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	U	X	X

Mnemonik:

	Takte	Buszyklen
SAR reg,1	2	-
SAR mem,1	15 + EA	2 (4)
SAR reg,CL	8 + 4/count	-
SAR mem,CL	20 + EA + 4/count	2 (4)

Kode:

1101 00cw	MD 111 R/M	(disp-lo)	(disp-hi)

c = 0 count = 1
c = 1 count = CL

SBB dest,src **Subtract With Borrow**

Operation: Der Quelloperand wird vom Zieloperanden subtrahiert. Von der Differenz wird 1 subtrahiert, falls C=1 ist. Der Zieloperand wird durch das Ergebnis ersetzt.

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

TYP 1:

Mnemonik:	Takte	Buszyklen
SBB reg,reg	3	-
SBB reg,mem	9 + EA	1 (2)
SBB mem,reg	16 + EA	2 (4)

Kode:

0001 00dw	MD REG R/M	(disp-lo)	(disp-hi)

TYP 2:

Mnemonik:	Takte	Buszyklen
SBB ac,data	4	-

Kode:

0001 010w	data-lo	(data-hi)

TYP 3:

Mnemonik:	Takte	Buszyklen
SBB reg,data	4	-
SBB mem,data	17 + EA	2 (4)

Kode:

1000 00sw	MD 010 R/M	(disp-lo)	(disp-hi)
		data-lo	(data-hi)

SCAS deststr **Scan String**
SCASB / SCASW

Operation:
1. Das Zielelement wird vom Akkumulatorinhalt subtrahiert. Die Operanden bleiben erhalten, die Flags werden aktualisiert.
2. SI und DI werden um 1 (w=0) oder 2 (w=1) erhöht (D=0) oder erniedrigt (D=1)

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

Mnemonik:	Takte	Buszyklen
SCAS deststr	15	1 (2)
SCASB / SCASW		
REP SCAS..	9 + 15/rep	1 (2)/rep

Kode:

1010 111w

SR: **(Prefix Byte)** **Segment Override**

Operation: Das implizite Segmentregister wird überschrieben.

Mnemonik:

	Takte	Buszyklen
CS:	2	-
DS:	2	-
ES:	2	-
SS:	2	-

Kode: 001 SR 110

SHR dest,count **Shift Arithmetic Right**

Operation:
1. Alle Bits werden um eine Stelle nach rechts verschoben. Das niederwertigste Bit wird in C geschoben. Die höchstwertige Bitposition wird auf 0 gesetzt.
2. Der Wert von count wird um 1 erniedrigt.
3. Die Operation wird wiederholt, solange count von 0 verschieden ist.

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	U	X	X

Mnemonik:

	Takte	Buszyklen
SHR reg,1	2	-
SHR mem,1	15 + EA	2 (4)
SHR reg,CL	8 + 4/count	-
SHR mem,CL	20 + EA + 4/count	2 (4)

Kode:

1101 00cw	MD 101 R/M	(disp-lo)	(disp-hi)

c = 0 count = 1
c = 1 count = CL

STC **Set Carry Flag**

Operation: Das Übertragsflag C wird auf 1 gesetzt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	1

Mnemonik:

	Takte	Buszyklen
STC	2	-

Kode: 1111 1001

STD **Set Direction Flag**

Operation: Das Richtungsflag D wird auf 1 gesetzt.

Flags:

O	D	I	T	S	Z	A	P	C
-	1	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	STD	2	-

Kode: 1111 1101

STI **Set Interrupt Flag**

Operation: Das Interruptflag I wird auf 1 gesetzt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	1	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	STI	2	-

Kode: 1111 1011

STOS deststr **Store String**
STOSB / STOSW

Operation:
1. Der Akkumulatorinhalt wird zum Zielelement übertragen.
2. SI und DI werden um 1 (w=0) oder 2 (w=1) erhöht (D=0) oder erniedrigt (D=1).

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

		Takte	Buszyklen
Mnemonik:	STOS deststr STOSB / STOSW	11	1 (2)
	REP STOS..	9 + 10/rep	1 (2)/rep

Kode: 1010 101w

SUB dest,src **Subtract**

Operation: Der Quelloperand wird vom Zieloperanden subtrahiert. Der Zieloperand wird durch das Ergebnis ersetzt.

Flags:

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

TYP 1: Mnemonik:		Takte	Buszyklen
	SUB reg,reg	3	-
	SUB reg,mem	9 + EA	1 (2)
	SUB mem,reg	16 + EA	2 (4)

Kode:				
	0010 10dw	MD REG R/M	(disp-lo)	(disp-hi)

TYP 2: Mnemonik:		Takte	Buszyklen
	SUB ac,data	4	-

Kode:			
	0010 110w	data-lo	(data-hi)

TYP 3: Mnemonik:		Takte	Buszyklen
	SUB reg,data	4	-
	SUB mem,data	17 + EA	2 (4)

Kode:				
	1000 00sw	MD 101 R/M	(disp-lo)	(disp-hi)
			data-lo	(data-hi)

TEST dest,src **Test**

Operation: Der Quelloperand wird mit dem Zieloperanden logisch "UND"-verknüpft. Die Operanden bleiben erhalten, die Flags werden aktualisiert.

Flags:

O	D	I	T	S	Z	A	P	C
0	-	-	-	X	X	U	X	0

TYP 1: Mnemonik:		Takte	Buszyklen
	TEST reg,reg	3	-
	TEST reg,mem	9 + EA	1 (2)

Kode:				
	1000 010w	MD REG R/M	(disp-lo)	(disp-hi)

TYP 2: Mnemonik:		Takte	Buszyklen
	TEST ac,data	4	-

Kode:			
	1010 100w	data-lo	(data-hi)

TYP 3: Mnemonik:		Takte	Buszyklen
	TEST reg,data	5	-
	TEST mem,data	11 + EA	1 (2)

Kode:				
	1111 011w	MD 000 R/M	(disp-lo)	(disp-hi)
			data-lo	(data-hi)

WAIT **Wait For Test**

Operation: Falls der TEST-Eingang nicht aktiv ist, wird der passive Zustand eingenommen, bis er aktiviert wird. Ist TEST aktiv, wird der nächste Befehl ausgeführt.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

	Takte	Buszyklen
Mnemonik: WAIT	3 + 5n	-

Kode: 1001 1011

XCHG dest,src **Exchange**

Operation: Der Quelloperand wird mit dem Zieloperanden vertauscht.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

TYP 1:

	Takte	Buszyklen
Mnemonik: XCHG reg,reg	4	-
XCHG mem,reg	17 + EA	2 (4)

Kode:

1000 011w	MD REG R/M	(disp-lo)	(disp-hi)

TYP 2:

	Takte	Buszyklen
Mnemonik: XCHG ac,reg16	3	-

Kode: 1001 OREG

XLAT (tab) **Translate**

Operation: Der Inhalt des AL-Registers wird durch den Inhalt einer Speicherzelle ersetzt, deren effektive Adresse aus der Summe der Inhalte von BX- und AL-Register gebildet wird.

Flags:

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

	Takte	Buszyklen
Mnemonik· XLAT (tab)	11	1

Kode: 1101 0111

XOR dest,src **Logical Exclusive OR**

Operation: Der Quelloperand wird mit dem Zieloperanden logisch "Exklusiv-ODER"-verknüpft. Der Zieloperand wird durch das Ergebnis ersetzt.

Flags:

O	D	I	T	S	Z	A	P	C
0	-	-	-	X	X	U	X	0

TYP 1:

Mnemonik:	Takte	Buszyklen
XOR reg,reg	3	-
XOR reg,mem	9 + EA	1 (2)
XOR mem,reg	16 + EA	2 (4)

Kode:

0011 00dw	MD REG R/M	(disp-lo)	(disp-hi)

TYP 2:

Mnemonik:	Takte	Buszyklen
XOR ac,data	4	-

Kode:

0011 010w	data-lo	(data-hi)

TYP 3:

Mnemonik:	Takte	Buszyklen
XOR reg,data	4	-
XOR mem,data	17 + EA	2 (4)

Kode:

1000 000w	MD 110 R/M	(disp-lo)	(disp-hi)
		data-lo	(data-hi)

Erläuterungen zu den Befehlskodes:

1. Allgemein gilt:

   w = 0 : dest, src = Byte
   w = 1 : dest, src = Wort
   d = 0 : dest = R/M
   d = 1 : dest = REG

2. Bei Arithmetik mit unmittelbaren Operanden gilt:

   sw = 00 : dest = Byte / src = Byte
   sw = 01 : dest = Wort / src = Wort
   sw = 10 : dest = Byte / src = Byte
   sw = 11 : dest = Wort / src = Byte

3. Der Adreßmodus (MD/REG/R/M) ist den Tafeln 3.4, 3.5, 3.6 zu entnehmen.

6. Programmierung des K1810WM86 in Assemblersprache

Programme für den K1810WM86 können in Assemblersprache oder in höheren Programmiersprachen geschrieben werden. Dieser Abschnitt gibt einen Überblick über die Grundlagen der Assemblerprogrammierung. Es wird die Struktur der Assemblersprache erläutert und die Vereinbarung von Daten, Segmenten und Prozeduren beschrieben.

6.1. Niveaus der Programmierung

Die Programmierung eines Mikroprozessors kann auf drei verschiedenen Niveaus erfolgen:

Maschinensprache
Die "Sprache" des Mikroprozessors besteht aus binär kodierten Maschinenbefehlen. Sie ist dem Menschen unverständlich, das Schreiben der Befehlskodelisten ist mühevoll und uneffektiv. Deshalb wird man davon nur noch in Ausnahmefällen Gebrauch machen, etwa bei Änderungen eines Speicherinhaltes. Dennoch muß jedes Programm, welches in irgendeiner Sprache geschrieben wurde, letztendlich in den Maschinenkode übertragen werden.

Assemblersprache
Diese Sprache ist ebenfalls maschinenorientiert, das heißt, sie verwendet genau den Befehlssatz des konkreten Mikroprozessors. Dadurch können dessen Fähigkeiten optimal ausgenutzt werden. Anstelle der Maschinenkodes werden Mnemoniks (Sprachsymbole) angewendet, welche bestimmte Befehle repräsentieren. Verschiedene zusätzliche Anweisungen (Direktiven) zur Vereinbarung von Daten, Prozeduren, Symbolen u.ä. gestatten eine flexible, verständliche und modulare Programmierung. Der Programmierer muß sich nicht mehr um alle Details kümmern, sondern ein Computerprogramm (der Assembler) übernimmt die Zuweisung von Speicherplätzen für Daten und Befehle, die Syntaxprüfung mit Fehlerprotokoll und Ausdruck der Programmliste und letztendlich die Übersetzung in die Maschinensprache.

Höhere Programmiersprachen
Diese problemorientierten Sprachen (z.B. FORTRAN, PASCAL, C, PL/M, BASIC) bieten viele Vorteile. Der Programmierer braucht keine Kenntnisse von der Prozessorarchitektur zu besitzen, so daß die Programmerstellung sehr effektiv erfolgen kann und die Programme vielseitig anwendbar sind. Auch hier muß ein Computerprogramm (der Compiler) helfen, das Programm in Maschinenkode umzusetzen. Dabei müssen allgemeine Befehlsstrukturen in die konkreten Prozessorbefehle umgesetzt werden, ein Vorgang, der sehr kompliziert ist. Deshalb können viele Probleme bezüglich Programmlaufzeiten und Speicherbedarf nicht optimal gelöst werden, ein Nachteil, der oft in Kauf genommen werden kann.

Die Tafel 6.1 zeigt einige Aspekte, die bei der Entscheidung für Assembler- oder höhere Sprache beachtet werden sollten.

Tafel 6.1. Vergleich von höherer Sprache und Assemblersprache

höhere Sprache	Assemblersprache
Vorteile	
- keine Kenntnisse von der Computerstruktur nötig - Programme auf verschiedene Computer übertragbar - Standardprogramme verfügbar - kurze Programmierzeiten	- optimierter Speicherbedarf - kürzeste Ausführungszeit - Ausnutzung aller prozessor-spezifischen Fähigkeiten - leicht erlernbare Syntax
Nachteile	
- nicht jede Sprache für jedes Problem geeignet - großer Speicherbedarf - größere Programmlaufzeiten	- Programme auf einen Prozessortyp zugeschnitten - uneffektive Formulierung komplexer Probleme - Hardwarekenntnisse nötig

Wann sollte die Assemblersprache benutzt werden? Ihre Anwendung kann dann empfohlen werden, wenn

- Echtzeitprobleme zu lösen sind,
- der Speicherbedarf minimiert werden soll,
- große Datenmengen transportiert werden müssen,
- Programme für viele Anwender des gleichen Computers verfügbar sein sollen.

6.2. Programmentwicklung

Die Programmentwicklung für den K1810WM86 erfolgt in mehreren Arbeitsschritten, die im Bild 6.1 schematisch dargestellt sind:

1. Erstellen einer Quelldatei mittels eines Textverarbeitungsprogramms. Jede in sich abgeschlossene Datei bildet einen Quellmodul.

2. Übersetzen der Quelldatei in eine Objektdatei; dazu wird ein Assembler- (oder Compiler-)programm angewendet. Die entstehenden Objektmodule sind nicht ausführbar, da ihnen noch keine absoluten Speicheradressen zugewiesen werden (sie sind verschieblich).

3. Erzeugen eines ausführbaren Programms. Ein Verbindeprogramm (Link & Locate) erzeugt dabei aus verschieblichen Objektdateien einen absoluten, lauffähigen Objektmodul, indem es verschiedene benötigte Module zusammensetzt und Speicheradressen zuweist.

Das wesentlichste Merkmal dieser Methode ist das Prinzip der modularen Programmierung. Die einzelnen in sich abgeschlossenen Teilprogramme (Module) können durch ein Bibliotheksprogramm verwaltet werden. Das Verbindeprogramm kann sowohl auf Objektdateien als auch Bibliotheksdateien zugreifen und daraus einen absoluten Objektmodul bilden.
Für beide Methoden, Assemblerprogrammierung und Programmierung in höheren Sprachen, ist der Ablauf bei der Programmentwicklung gleich. Wenn Assembler und Compiler gleiche Objektformate erzeugen, können in verschiedenen Sprachen geschriebene Programmodule miteinander kombiniert werden.
Die zur Programmentwicklung benötigten Dienstprogramme sind abhängig von dem Betriebssystem, welches auf dem als Entwicklungssystem benutzten Computer installiert ist. Daher soll hier auf eine Beschreibung derselben verzichtet und der Nutzer auf entsprechende Gerätedokumentationen verwiesen werden.

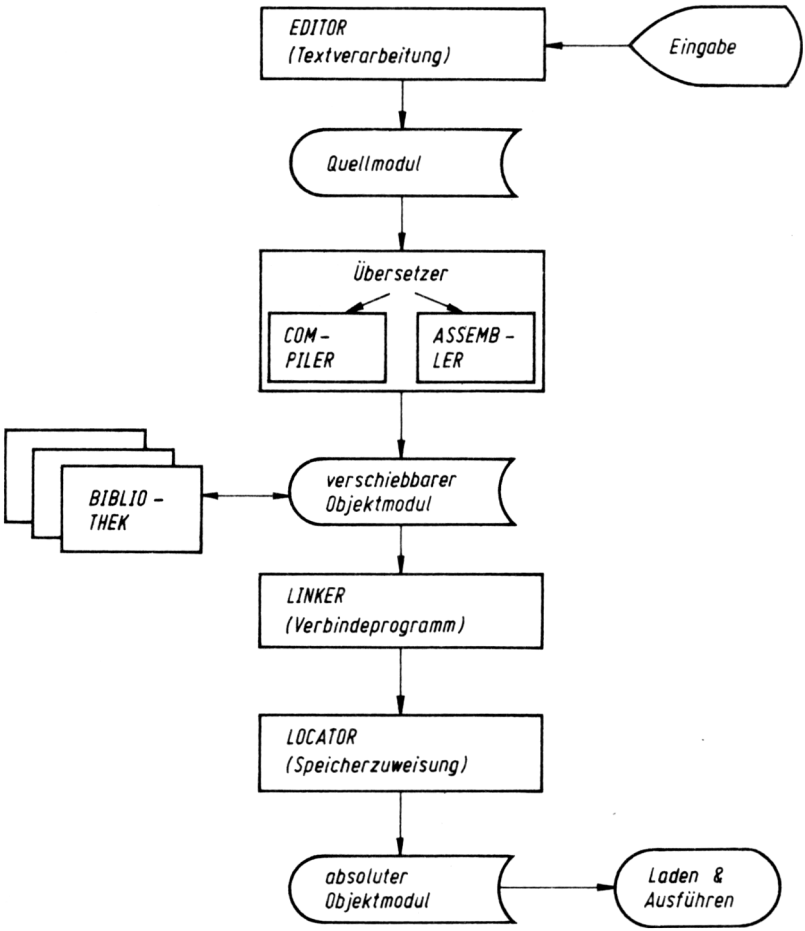


Bild 6.1. Prinzip der Programmentwicklung

6.3. Struktur der Assemblersprache

6.3.1. Assembleranweisungen

Eine Anweisung in einem Assemblerprogramm kann ein Befehl oder eine Direktive sein. Befehle bilden das eigentliche Programm, während Direktiven der Steuerung des Assemblers dienen.
An die Assembleranweisungen werden keine speziellen Formatforderungen gestellt. Der Assembler identifiziert jede Anweisung anhand ihrer Struktur. Jeder Anweisung ist ein Sprachsymbol - ein Mnemonik - zugeordnet.
Verschiedene Maschinenbefehle, die analoge Operationen mit unterschiedlichen Operanden ausführen, werden durch identische Mnemoniks bezeichnet. Dadurch reduziert sich der aus etwa 300 Maschinenbefehlen bestehende Befehlssatz des K1810WM86 auf rund 100 Assemblerbefehle. Die Programmierung wird dadurch überschaubarer.

Beispielsweise kann der Befehl

```
MOV destination,source
```

je nach Operandentyp 28 verschiedene Datentransferbefehle von 2 bis 6 Byte Länge bewirken.

6.3.2. Zahlen und Zeichen

Die Assemblersprache verarbeitet Ziffern, Groß- und Kleinbuchstaben und Sonderzeichen. Zahlen können als ganze Zahlen im Binär-, Oktal-, Dezimal- oder Hexadezimalformat dargestellt werden. Für Arithmetikoperationen mit dem Numerikcoprozessor K1810WM87 sind außerdem die Formate Real und BCD möglich. Als Standard wird Dezimal angenommen.
Zeichen und Zeichenketten (Strings) müssen in Anführungszeichen gesetzt werden, wenn sie als Operanden verwendet werden. Dabei werden Groß- und Kleinbuchstaben unterschieden.

```
Beispiele:   00101101B ;BINAERZAHL
             1234Q     ;OKTALZAHL
             1987D     ;DEZIMALZAHL
             100       ;DEZIMALZAHL, MUSS NICHT VEREINBART SEIN
             0C8H      ;HEXADEZIMALZAHL, BEGINNT STETS MIT ZIFFER
             -73.28    ;REALZAHL
             3.45E-6   ;REALZAHL MIT EXPONENT
             -9876543  ;BCD-ZAHL, MAXIMAL 18 STELLEN
             "A"       ;ZEICHEN
             "Name:"   ;STRING
```

Neben Zahlen und Zeichen werden auch Namen angewendet, zum Beispiel für Marken. Dabei ist darauf zu achten, daß diese nicht mit reservierten Namen (Mnemoniks, Registerbezeichnungen u.ä.) übereinstimmen.

6.3.3. Assemblerbefehle

Assemblerbefehle sind ausführbare Anweisungen. Der Assembler generiert aus jedem Befehl einen speziellen Maschinenbefehl. (Assemblerbefehle bilden also das Programm, welches das Zielsystem ausführen soll.)
Die Assemblerbefehle werden in folgender Form geschrieben:

(MARKE:) MNEMONIK (ZIELOPERAND)(,QUELLOPERAND) (;KOMMENTAR)

Alle Angaben in Klammern können vorhanden sein oder ganz oder teilweise entfallen (die Klammern werden nicht mitgeschrieben). Die einzelnen Felder haben folgende Bedeutung:

MARKE (Label)
Dieses Feld bezeichnet den Speicherplatz, auf dem der Befehl zu finden ist. Marken werden in der Form "NAME DOPPELPUNKT" angegeben. Sie können von Sprungbefehlen als Operand benutzt werden, der Assembler ermittelt aus dem Namen die Zieladresse des Sprungs.

MNEMONIK
Dieses Feld bezeichnet den Befehlstyp, dessen Kode generiert werden soll. Befehle können darüber hinaus mit einem Präfix versehen sein, der besondere Eigenschaften des Befehls definiert (z.B. LOCK, REP). Dann erhält dieses Feld folgende Form:

PRÄFIX MNEMONIK.

ZIELOPERAND, QUELLOPERAND (Destination, Source)
Bestimmte Befehle fordern keinen, einen oder zwei Operanden. Zwei Operanden werden durch "KOMMA" getrennt und das Ziel zuerst geschrieben, gefolgt vom Quelloperanden. Der Operandentyp (Register, Konstante, Variable, Marke) bestimmt die Adressierungsart, die der Befehl anwendet.

KOMMENTAR
Das Kommentarfeld beginnt mit "SEMIKOLON" und dient der Erläuterung des Befehls im konkreten Programm. Ein gutes Programm ist stets mit verständlichen Kommentaren versehen.

6.3.4. Assemblerdirektiven

Direktiven generieren selbst keine Maschinenbefehle, sondern steuern den Assembler. (Sie werden also nur auf dem Entwicklungssystem ausgeführt und heißen deshalb auch Pseudobefehle.) Sie sind erforderlich, um Speicherplätze zu reservieren, Operandentypen zu vereinbaren, Adressen zuzuweisen, die Dokumentationserstellung (Listings) zu steuern u.ä.
Direktiven werden in folgender Form geschrieben:

(NAME) MNEMONIK (OPERANDEN) (;KOMMENTAR)

Die Angaben in Klammern können entfallen, wenn die konkrete Anweisung sie nicht verlangt. Hinter NAME steht kein Doppelpunkt; mehrere Operanden werden durch Kommas getrennt.

Direktiven sind assemblerspezifische Anweisungen und können bei verschiedenen Assemblern unterschiedlich sein. Der Programmierer muß sich also mit den konkreten Assemblerdokumentationen vertraut machen. Zum besseren Verständnis sollen im folgenden die wichtigsten Direktiven kurz erläutert werden. (Es muß darauf hingewiesen werden, daß viele Assembler weit mehr Direktiven beinhalten als hier aufgeführt werden.)

6.4. Vereinbarung von Daten, Segmenten und Prozeduren

6.4.1. Symbolzuweisung

Die Symbolzuweisung ist eine allgemeine Direktive zur Vereinbarung von Symbolen (Namen), die an unterschiedlichen Stellen im Programm anstelle bestimmter Ausdrücke verwendet werden sollen. Eine solche Direktive hat den Vorteil, daß bei der Anwendung eines Symbols nur die Direktive geändert werden muß, wenn dem Symbol ein anderer Ausdruck zugewiesen werden soll.
Die Symbolzuweisung hat die Form:

```
NAME EQU AUSDRUCK
```

Die EQU-Answeisung weist einem Ausdruck einen symbolischen Namen zu, der im Programm an dessen Stelle verwendet wird.

Der Ausdruck kann beispielsweise sein:

- ein Wert (Konstante)
- ein logischer oder arithmetischer Ausdruck
- ein Register
- eine Adresse
- ein Befehl

Beispiele:

```
CONST     EQU 100      ;KONSTANTE MIT DEM WERT 100
NAME      EQU "Name"   ;ASCII-STRING
SEGMENT_1 EQU 2F00H    ;SEGMENTBASIS
```

6.4.2. Datenvereinbarungen

Datenvereinbarungen dienen dem Reservieren von Speicherplätzen, der Typzuordnung von Variablen (z.B. Byte, Wort) und der Initialisierung von Variablen mit bestimmten Werten. Sie haben die Form:

```
NAME TYP AUSDRUCK          (TYP = DB / DW / DD / DQ / DT)
```

Jeder Variablen wird ein Name, ein Typ und ein Ausdruck zugewie-

sen. Soll nur Speicherplatz reserviert werden, kann auch der Name entfallen.
Der Typ kann sein:

```
DB   : Byte
DW   : Wort
DD   : Doppelwort (Pointer, Short-Integer oder Short-Real)
DQ   : Vierfachwort (Long-Integer oder Long-Real)
DT   : Zehn Bytes  (BCD, Temporary-Real)
```

(Diese Typen erlauben u.a. die Vereinbarung von Datenformaten für den Numerikcoprozessor K1810WM87, die im Abschnitt 7 beschrieben sind.)
Der Ausdruck kann beispielsweise sein:

- ein Wert, mit dem der reservierte Speicherplatz initialisiert werden soll,
- ?, wenn keine Initialisierung erforderlich ist,
- ein Name, der in einer anderen Direktive bereits vereinbart wurde.

Jeder Variablen wird vom Assembler eine Zahl von Attributen zugeordnet (siehe auch Tafel 6.2):

- OFFSET (Offsetadresse des ersten Bytes der Variablen)
- TYP (1=Byte, 2=Wort, 4=Doppelwort)
- LENGTH (Zahl der Elemente)
- SIZE (Zahl von Bytes, d.h. TYP x LENGTH)

Es können auch ganze Felder (Arrays) mit einem Variablennamen vereinbart werden; dazu dient die DUP-Anweisung.

Beispiele:

```
ALPHA     DB   ?
BETA      DW   1000
GAMMA     DB   "A"
POINTER   DD   SEGMENT_1
DELTA     DW   10 DUP (?)     ;RESERVIERT 10 WORTE
TABLE     DB   10H,80H,0F0H,0F8H,0FCH
          DW   256 DUP (?)
```

Tafel 6.2. Attribute der Speichervariablen

NAME	TYP	OFFSET	LENGTH	SIZE
ALPHA	1	0	1	1
BETA	2	1	1	2
GAMMA	1	3	1	1
POINTER	4	4	1	4
DELTA	2	8	10	20
TABLE	1	28	5	5

Der Assembler kann diese Attribute als Operanden für Befehle verarbeiten. Beispielsweise würde dann der Befehl

```
MOV BX,OFFSET ALPHA
```

das Register BX mit der Offsetadresse der Variablen ALPHA laden, nicht mit deren Wert.

Soll der Typ einer Variablen geändert werden oder wurde kein Typ vereinbart (etwa bei indirekter Speicheradressierung), so kann direkt in einem Assemblerbefehl folgende Direktive angewendet werden:

```
TYP PTR Operand          (TYP = BYTE / WORT / DWORT)
```

Beispiele:

```
MOV WORD PTR [BX], 10H
MOV CL,BYTE PTR DELTA[SI]
```

Oft ist es sinnvoll, Daten vom Typ DW, DD, DQ oder DT auf geraden Adressen beginnen zu lassen. Dann kann die Direktive EVEN verwendet werden. Diese organisiert, daß die nachfolgende Datenvereinbarung auf eine gerade Adresse justiert wird.

Beispiel:

```
DW 10 DUP (?)        ;RESERVIERT 10 WORTE
DB 0FFH
EVEN                 ;EIN BYTE WIRD UEBERSPRUNGEN
DW 1000
```

6.4.3. Segmentvereinbarungen

Der Assembler muß wissen, welche logischen Speichersegmente den Programmen und Daten zugewiesen und wo diese Segmente schließlich im Speicher angeordnet werden sollen. Die Segmentzuweisung hat folgende Form:

```
NAME   SEGMENT
.
.
NAME   ENDS
```

Diese Direktive dient der Vereinbarung logischer Segmente. Zwischen den Anweisungen SEGMENT und ENDS können Datenvereinbarungen oder Befehle stehen. Es ist möglich, das gesamte Programm einschließlich aller Daten in ein Segment zu schreiben. Üblicherweise werden aber für Kode, Daten und Stapel unterschiedliche Segmente definiert.
Die Direktive

```
ASSUME SR:NAME                (SR = CS / DS / ES / SS)
```

weist jedem logischen Segment ein Segmentregister zu.

Das folgende Beispiel soll den Aufbau eines Assemblerprogramms demonstrieren:

```
DATA      SEGMENT
          VARIABLE_1  DB ?
     ;HIER FOLGEN ALLE DATENVEREINBARUNGEN
DATA      ENDS
STACK     SEGMENT
          DW 100 DUP (?) ;RESERVIERT 100 WORTE STACK
STACK     ENDS
CODE      SEGMENT
ASSUME CS:CODE, DS:DATA, ES:DATA, SS:STACK
     ;DS UND ES UEBERLAPPEN VOLLSTAENDIG

INIT:     MOV AX,DATA
          MOV DS,AX
          MOV ES,AX
          MOV AX,STACK
          MOV SS,AX
     ;CS BEHAELT DEN MIT RESET INITIALISIERTEN WERT
     ;ALLE SEGMENTE SIND JETZT ADRESSIERBAR
     ;HIER FOLGT DER PROGRAMMKODE
CODE      ENDS
          END  INIT
```

Die Direktive END MARKE weist nun den Assembler an, die Assemblierung zu beenden und vereinbart eine Anfangsadresse für das Programm, welches mit der Marke INIT beginnt. Damit sind alle Vorraussetzungen gegeben, um dem Programm und allen Speicherdaten physikalische Adressen zuzuweisen.

6.4.4. Prozedurvereinbarung

Prozeduren sind Programme innerhalb eines Moduls, aus denen das gesamte auszuführende Programm schließlich zusammengesetzt wird. Eine Prozedurvereinbarung hat die Form:

```
NAME   PROC   TYP        (TYP = NEAR / FAR)
.
.
NAME   ENDP
```

Diese Direktive vereinbart Prozeduren (Unterprogramme). Die Prozedurnamen können ähnlich wie Marken als Operanden in Assemblerbefehlen benutzt werden (z.B. CALL procedur).
Der Typ der Prozedur bestimmt den Typ der CALL- und RET-Befehle. Eine fehlende Typangabe wird als NEAR interpretiert.
Um Fehler zu vermeiden, wenn Marken vor ihrer Vereinbarung verwendet werden, sollte der PTR-Operator angewendet werden:

Beispiel: CALL FAR PTR UPROG

6.4.5. Globale Symbole

Ein konkretes Programm zur Lösung einer bestimmten Aufgabe wird selten aus einem einzigen Modul bestehen. Die Aufteilung in mehrere Module bietet verschiedene Vorteile. Jeder Modul kann einzeln übersetzt und getestet werden, Module können ausgetauscht und in unterschiedliche Programme einbezogen werden, die Übersichtlichkeit und nicht zuletzt die Verwaltung werden erleichtert.
Die Verbindung einzelner Module zu einem ausführbaren Programm erfordert verschiedene Anweisungen, welche die Verflechtungen zwischen den Modulen regeln. Jeder Modul kann Marken, Variable oder Konstanten aus anderen Modulen verwenden. Zur Vereinbarung globaler Symbole dienen folgende Direktiven:

```
PUBLIC    SYMBOL,...,SYMBOL
EXTRN     SYMBOL:TYP,...,SYMBOL:TYP
```

Symbole können die Namen von Marken, Variablen oder Konstanten sein.
Die PUBLIC-Anweisung wird in dem Modul verwendet, in dem die Symbole vereinbart wurden. Damit sind diese zur Benutzung durch andere Module freigegeben.
Die EXTRN-Anweisung muß in den Modulen stehen, welche auf nicht vereinbarte Symbole zugreifen. Diese Symbole müssen dort, wo sie vereinbart wurden, als PUBLIC definiert sein. Damit der Assembler den Typ der externen Symbole kennt, muß dieser angegeben werden (zum Beispiel BYTE oder WORD bei Variablen, NEAR oder FAR bei Marken, ABS bei Konstanten).

Beispiele:

```
PUBLIC   ALPHA, BETA, BEGIN, UPROG
ALPHA    EQU    "X"
BETA     DW     0
BEGIN    LABEL  FAR
UPROG    PROC   FAR

EXTRN    ALPHA:ABS, BEGIN:FAR
EXTRN    BETA:WORD, UPROG:FAR
```

Darüber hinaus sind weitere Vereinbarungen in Abhängigkeit vom konkreten Assembler- bzw. Verbindeprogramm möglich, die den Komfort der Programmentwicklung bestimmen.

7. Numerikcoprozessor K1810WM87

Der K1810WM87 ist ein leistungsfähiger Arithmetikprozessor. Gemeinsam mit einem K1810WM86 kann er vorteilhaft für rechenintensive Anwendungen eingesetzt werden - wie Computergrafik, wissenschaftliche Berechnungen oder Steuer- und Regelaufgaben.

7.1. Aufbau und Arbeitsweise des K1810WM87

7.1.1. Übersicht

Der K1810WM87 ist ein Coprozessor, dessen Hardware für den Anwender eines K1810WM86-Systems unsichtbar bleibt. Deshalb soll hier nur ein kurzer Überblick gegeben werden.
Der K1810WM87 arbeitet unmittelbar mit einem K1810WM86 (Host CPU) zusammen, ohne daß er als separate Einheit - wie beispielsweise ein Ein-/Ausgabegerät - behandelt werden muß. Dem Anwender steht gegenüber einem einzelnen K1810WM86 ein erweiterter Befehls- und Registersatz zur Verfügung, und er kann mit einigen neuen Datentypen operieren. Existierende Software wird durch das Hinzufügen des Coprozessors nicht beeinflußt.
Der K1810WM87 wurde speziell für die schnelle Ausführung arithmetischer Operationen einschließlich transzendenter Funktionen entwickelt. Die Tafel 7.1 zeigt eine Gegenüberstellung der Ausführungszeiten typischer Operationen, die unmittelbar mit einem K1810WM87 oder mit K1810WM86-Prozeduren ausgeführt werden.

Tafel 7.1. Vergleich einiger Gleitkommaoperationen

Operation	Ausführungszeit in µs	
	K1810WM87	K1810WM86 (Emulation)
Multiplikation,		
einfache Genauigkeit	19	1600
doppelte Genauigkeit	27	2100
Division, einfache Genauigkeit	39	3200
Addition:	17	1600
Quadratwurzel:	36	19600
Tangens:	90	13000
Exponentialfunktion:	100	17100

Es soll angemerkt werden, daß der K1810WM87 höchste Genauigkeit bei der Umsetzung arithmetischer Algorithmen bietet und in dieser Hinsicht konventionellen Gleitkomma-Prozessoren und Minicomputern überlegen ist. Es können 64-Bit-Binärzahlen und Dezimalzahlen mit 18 Stellen exakt verarbeitet werden. Da das interne Zahlenformat noch größere Zahlen zuläßt, treten auch dann keine Ungenauigkeiten auf, wenn Zwischenergebnisse diese Bereiche überschreiten.

7.1.2. Hardware-Interface

Der K1810WM87 ist in der Lage, Befehle parallel zum K1810WM86 zu dekodieren und auszuführen. Dies ist möglich durch ein einheitliches Interface und spezielle Mechanismen zur Synchronisation. Bild 7.1 zeigt die Anschlußbelegung des K1810WM87. Die meisten Signale sind mit denen des K1810WM86 identisch. Eine spezielle Bedeutung haben folgende Signale:

BUSY signalisiert, daß der K1810WM87 eine Operation ausführt. Dieses Signal wird mit dem /TEST-Eingang des K1810WM86 verbunden, um eine Synchronisation zu ermöglichen.

INT signalisiert eine Interruptanforderung des K1810WM87. Dieses Signal wird mit dem Interruptcontroller des K1810WM86 verbunden.

/RQ/GT0 wird mit einem der /RQ/GT-Anschlüsse des K1810WM86 verbunden, um die Buszugriffe zu steuern.
(An den zweiten /RQ/GT-Anschluß kann ein weiterer Coprozessor angeschlossen werden.)

Alle anderen Anschlüsse werden mit den entsprechenden Anschlüssen des K1810WM86 verbunden (Bild 7.2). Es sind keine weiteren Hardwarekomponenten erforderlich.

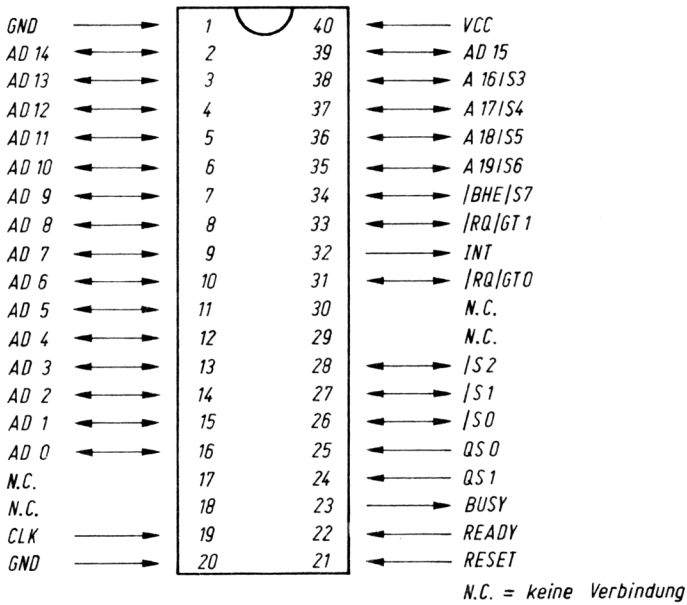


Bild 7.1. Anschlußbelegung des K1810WM87

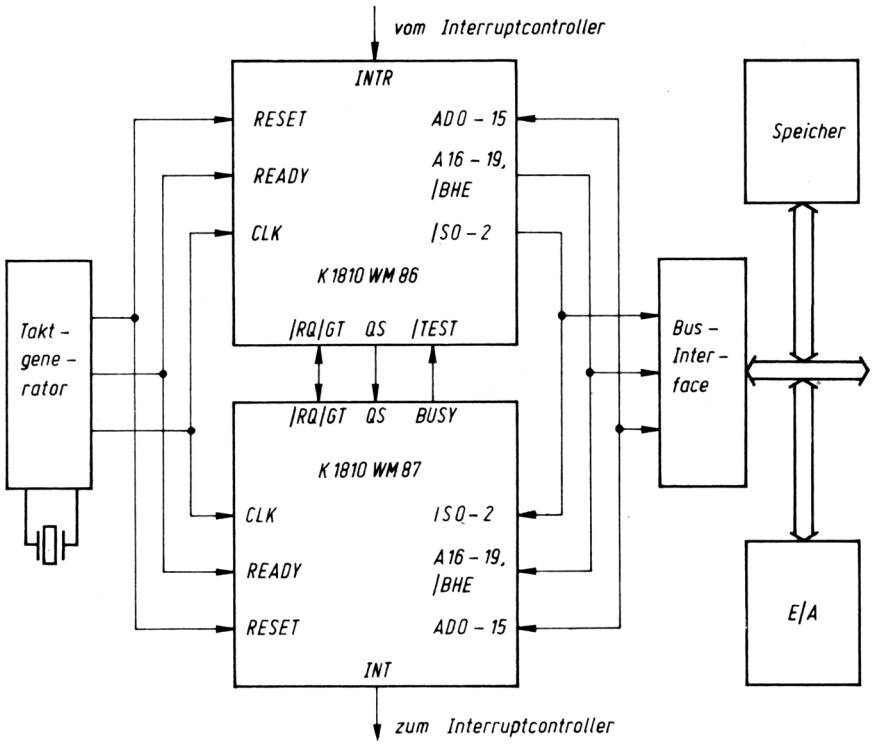


Bild 7.2. Arithmetikprozessor mit K1810WM87

7.1 3. Software-Interface

Der Programmierer kann das System K1810WM86/87 als eine einzige Maschine betrachten. Befehle für den K1810WM87 erscheinen in einem sequentiellen Befehlsstrom mit K1810WM86-Befehlen gemischt. Die Coprozessorarchitektur beachtet automatisch die Koordinierung zwischen den beiden Prozessoren.
Der K1810WM87 beobachtet im passiven Zustand alle Busleitungen. Erkennt er, daß der K1810WM86 einen ESC-Kode als Befehl einliest, so interpretiert er diesen als ausführbaren Befehl. Alle anderen Befehle ignoriert er, so wie der K1810WM86 den ESC-Befehl ignoriert. Beide Prozessoren können Befehle parallel dekodieren und ausführen (siehe auch Abschn. 7.2.3).

Programme können wie für einen einzelnen K1810WM86 geschrieben werden. Obwohl Compiler existieren, welche die Anwesenheit eines Numerikcoprozessors berücksichtigen, erreicht man die volle Leistungsfähigkeit mit Programmen in Assemblersprache. Der K1810WM86-Assembler beinhaltet Direktiven zur Vereinbarung aller Datentypen und Mnemoniks für alle zusätzlichen Maschinenbefehle.

Dem Programmierer stehen für ein K1810WM86/87-System folgende Ressourcen zur Verfügung:

- 160 Befehle,
- 12 Datentypen,
- 4 Segmente,
- 8 allgemeine Datenregister (16 Bit),
- 8 Gleitkommaregister (80 Bit).

Darüber hinaus besteht die Möglichkeit, mit einem Software-Emulator Programme für den K1810WM87 auf einem K1810WM86 laufen zu lassen. Der Emulator ist ein Programm, das alle ESC-Befehle ausfiltert und durch Unterprogramme für den K1810WM86 ersetzt. Das Quellprogramm unterscheidet sich dabei nicht. Erst im Verbindeprogramm muß angegeben werden, ob ein K1810WM87 vorhanden ist oder emuliert werden soll. Dadurch ist es möglich, hardwareunabhängige Programme zu schreiben und zu testen. Der Numerikcoprozessor kann dann zu jeder Zeit problemlos nachgerüstet werden, die Programmodule müssen lediglich neu verbunden werden.

7.1.4. Datentypen und Formate

Der K1810WM87 führt alle internen Berechnungen mit Gleitkommazahlen durch. Diese werden im binären System wie folgt dargestellt:

$$(-1)^S \times I,FF...FF \times 2^{EE...EE}$$

S - Vorzeichen (S = 0: positiv, S = 1: negativ)
I - Integerbit (I = 1 bei normierten Zahlen)
FF...FF - Fraktionsbits
EE...EE - Exponentbits

Die Größe I,FF...FF wird als Mantisse bezeichnet. Beim K1810WM87 wird auch der Begriff "Signifikand" angewendet. Die Anzahl der Fraktionsbits bestimmt die Genauigkeit der Zahl (Anzahl der signifikanten Stellen). Durch Normierung liegt der Signifikand immer im Bereich zwischen 1 und 2. Dadurch werden führende Nullen eliminiert und die Genauigkeit für kleine Zahlen erhöht. Das Integerbit ist dabei stets 1 und muß nicht dargestellt werden. Der Exponent gibt an, wo der wirkliche Binärpunkt liegt. Positive Exponenten verschieben ihn nach rechts, negative nach links (Anzahl der führenden Nullen). Die Anzahl der Exponentbits bestimmt die darstellbaren Größenbereiche.

Für unterschiedliche Anwendungen und Anforderungen kann der K1810WM87 sieben verschiedene Datenformate verarbeiten (Tafel 7.2 und Bild 7.3). Diese können in drei Klassen eingeteilt werden:

Ganze Binärzahlen (Binary Integer)
Diese im folgenden Integerzahlen genannten Zahlen werden im Zweierkomplement dargestellt. Das höchstwertige Bit ist das Vorzeichen.

Gepackte BCD-Zahlen (Packed Decimal)
Jedes Dezimaldigit wird durch 4 Bit dargestellt (BCD-Kode). Das höchstwertige Bit ist das Vorzeichen. Sie werden im 80-Bit-Format dargestellt (72 signifikante Bits, die Bits 72 - 78 sind ohne Bedeutung und werden auf 0 gesetzt).

Gleitkommazahlen (Real)
Diese, im folgenden als Real-Zahlen bezeichneten Zahlen werden in der eingangs beschriebenen Form dargestellt. Das höchstwertige Bit ist das Vorzeichen, dann folgt das Exponentfeld und anschließend das Signifikandfeld.

Tafel 7.2. Datenformate des K1810WM87

Datentyp	Bits	Zahlenbereich
Word Integer	16	-32768 bis $+32767$
Short Integer	32	-2×10^{9} bis $+2 \times 10^{9}$
Long Integer	64	-9×10^{18} bis $+9 \times 10^{18}$
BCD	80	$-10^{18} - 1$ bis $+10^{18} - 1$
Short Real	32	$\pm 8,4 \times 10^{-37}$ bis $\pm 3,4 \times 10^{38}$
Long Real	64	$\pm 4,2 \times 10^{-307}$ bis $\pm 1,7 \times 10^{308}$
Temporary Real	80	$\pm 3,4 \times 10^{-4932}$ bis $\pm 1,2 \times 10^{4932}$

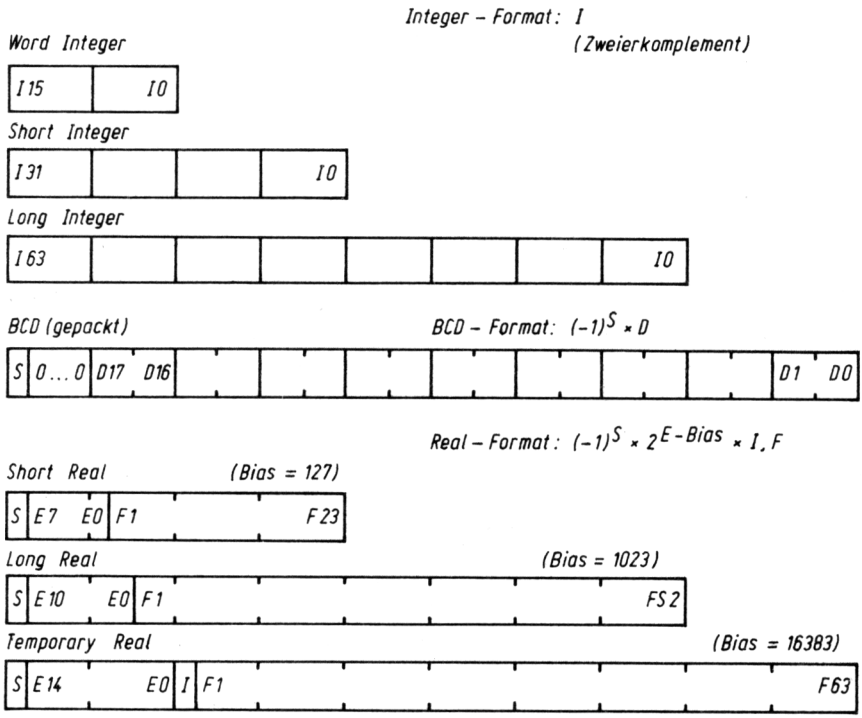


Bild 7.3. Zahlenformate

Im Short- und Long-Real-Format wird das Integerbit implizit angenommen; im Temporary-Real-Format ist es explizit enthalten (Bit 63).
Aus praktischen Gründen arbeitet der K1810WM87 nicht mit dem wirklichen Exponenten, sondern addiert stets einen konstanten Wert hinzu (Bias). Dieser ist so gewählt, daß das Exponentfeld nur positive Werte enthält. Für die verschiedenen Formate ist der Bias dementsprechend unterschiedlich:

Short Real	: 8 Bit Exponent;	Bias	127	(7FH)
Long Real	: 11 Bit Exponent;	Bias	1023	(3FFH)
Temporary Real	: 15 Bit Exponent;	Bias	16383	(3FFFH)

Das bedeutet zum Beispiel, daß ein Exponent von +7 im Long-Real-Format als 4006H dargestellt wird.

Die Formate Integer, BCD sowie Short und Long Real wendet der K1810WM87 nur bei Speicheroperationen an. Intern arbeitet er ausschließlich im Temporary-Real-Format. Das bedeutet, daß beim Laden einer Integerzahl diese automatisch konvertiert wird. Andererseits wird beim Abspeichern eine Zahl ebenfalls in das geforderte Format konvertiert und gegebenenfalls gerundet (ohne Verlust an Genauigkeit). Dadurch ist es möglich, daß auch dann noch genaue Ergebnisse erhalten werden, wenn Zwischenergebnisse das geforderte Format überschreiten. Das Temporary-Real-Format kann auch für Speicheroperanden vereinbart werden, sollte aber Zwischenergebnissen oder hochgenauen Konstanten vorbehalten bleiben.

7.1.5. Register

Der K1810WM87 enthält zur Ausführung seiner Operationen einen Satz Gleitkommaregister sowie verschiedene Status- und Steuerregister (Bild 7.4).

Stapelregister:
Die 8 Gleitkommaregister sind 80 Bit breit. Sie nehmen die Operanden für die arithmetischen Operationen auf. Jedes dieser Register ist in ein 64 Bit breites Signifikandfeld, ein 15 Bit breites Exponentfeld und ein Vorzeichenbit geteilt.
Dieser Registersatz ist als Stapel organisiert. Das Register, welches jeweils zuletzt geladen wurde, bildet die aktuelle Stapelspitze (Top Of Stack). Der Stapelzeiger im Statuswort (TOS-Feld) gibt an, welches der Register die momentane Stapelspitze bildet. Soll ein Operand geladen werden, wird der Stapelzeiger um 1 erniedrigt und der Wert in das Register (die neue Stapelspitze) eingetragen. Umgekehrt bewirken andere Befehle, daß der Inhalt der Stapelspitze ausgegeben wird und der Stapelzeiger anschließend erhöht wird. Der Stapel wächst also in Richtung niederer Registeradressen und ist nach dem Prinzip "zuletzt hinein, zuerst heraus" organisiert.

Steuerwort:
Dieses 16-Bit-Register muß über den Speicher geladen werden, um verschiedene Betriebsarten für unterschiedliche Anwendungen zu programmieren. Es hat folgendes Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	R	R	R	IC	RC		PC		ID	R	PM	UM	OM	ZM	DM	IM

R : reserviert (ohne Bedeutung)

IC : Infinitätsmodell (Infinity Control)
0 = projektiv ; der Wert Unendlich ist vorzeichenlos.
1 = affin ; der Wert Unendlich ist vorzeichenbehaftet.

RC : Rundungsverfahren (Rounding Control)
00 = runde zum nächstliegenden Wert
01 = runde abwärts
10 = runde aufwärts
11 = schneide in Richtung 0 ab ("chop")

Das RC-Feld bestimmt den angewendeten Rundungsmechanismus. Dazu sucht der K1810WM87 stets die zwei exakt darstellbaren benachbarten Werte und rundet zu einem der beiden, je nach ausgewähltem Modus. RC = 00 ist das für die meisten Anwendungen günstigste Verfahren.

PC : Genauigkeitskontrolle (Precision Control)
00 = 24 Bit
01 = reserviert
10 = 53 Bit
11 = 64 Bit

Das PC-Feld bestimmt die Genauigkeit des Ergebnisses. Der Wert gibt an, wieviel signifikante Bits benutzt werden. (Obwohl intern im Temporary-Real-Format gerechnet wird, kann die Genauigkeit von vornherein auf Short- oder Long-Real-Format eingeschränkt werden.) Die Operanden werden gerundet und die nichtbenutzten niederwertigsten Bits auf 0 gesetzt.

ID : Interruptmaske (Interrupt Enable Mask)
0 = Interrupts erlaubt
1 = Interrupts verboten

Exception-Masken:
0 = Exception nicht maskiert
1 = Exception maskiert

PM : Genauigkeit (Precision Mask)
UM : Unterschreitung (Underflow Mask)
OM : Überlauf (Overflow Mask)
ZM : Division durch Null (Zero Divide Mask)
DM : denormaler Operand (Denormal Mask)
IM : ungültige Operation (Invalid Operation Mask)

Statuswort:
Das 16-Bit-Statusregister spiegelt den Zustand des K1810WM87 als Ergebnis einer Operation wieder. Es kann in den Speicher abgebildet und dort über den K1810WM86 getestet werden. Es hat folgendes Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	B	C3	T	O	S	C2	C1	C0	IR	R	P	U	O	Z	D	I

R : Reserviert (ohne Bedeutung)
B : Ausführungseinheit besetzt (Busy)
C3...C0 : Ergebniskode (Condition Code)
TOS : Stapelzeiger (Top Of Stack Pointer)
IR : Interruptanforderung (Interrupt Pending Request)

Exception-Flags:

P : Genauigkeit (Precision)
U : Unterschreitung (Underflow)
O : Überlauf (Overflow)
Z : Division durch Null (Zero Divide)
D : denormaler Operand (Denormal Operand)
I : ungültige Operation (Invalid Operation)

Das Bit B reflektiert den Zustand der BUSY-Leitung. Es ist auf 1 gesetzt, wenn der K1810WM87 eine laufende Operation noch nicht beendet hat.
Der Stapelzeiger TOS enthält den Binärkode der aktuellen Stapelspitze.
Der Ergebniskode C3 bis C0 reflektiert das Ergebnis von Vergleichsoperationen (s. Abschn. 7.3.4).
Das IR-Bit zeigt eine nichtbearbeitete Interruptanforderung an. Es ist auf 1 gesetzt, wenn eine Interruptanforderung vorliegt.
Die Exception-Flags sind auf 1 gesetzt, wenn bei der Ausführung eines Befehls eine Fehlerbedingung aufgetreten ist.

TAG-Wort:
Dieses 16-Bit-Register wirkt als Kennzeichnung für den Zustand der 8 Stapelregister. Es wird nur unter besonderen Umständen nötig sein, seinen Inhalt zu kennen. Es kann nur mit den FSAVE- oder FSTENV-Befehlen im Speicher abgebildet werden. Es hat folgendes Format:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	T(7)		T(6)		T(5)		T(4)		T(3)		T(2)		T(1)		T(0)	

T(i) : 00 = gültiger Wert
01 = Null
10 = spezieller Wert (Denormal, Unendlich, NAN)
11 = frei

Exception Pointer:
Der K1810WM87 enthält zwei Doppelwortzeiger. Im Befehlszeiger (IP) werden stets die vollständige Adresse des aktuellen Befehls und der Befehlskode zwischengespeichert. Im Operandenzeiger (OP) wird die vollständige Adresse eines Speicheroperanden zwischengespeichert. Beim Auftreten einer unmaskierten Exception kann die Interruptprozedur diese beiden Zeigerregister im Speicher abbilden, um Informationen über den fehlerhaften Befehl und dessen Operanden zu gewinnen. Die Pointer haben folgendes Format:

Bit	15 14 13 12	11	10	9	8	7	6	5	4	3	2	1	0
	OP19 - 16	0	0	0	0	0	0	0	0	0	0	0	0
	OP15 - 0												
	IP19 - 16	0	CODE 10 - 0										
	IP15 - 0												

IP19 - 0 : 20-Bit-Adresse des aktuellen Befehls
CODE10 - 0 : 11-Bit-Operationskode (die höchstwertigen Bits 15-11 sind immer 11011B)
OP19 - 0 : 20-Bit-Operandenadresse

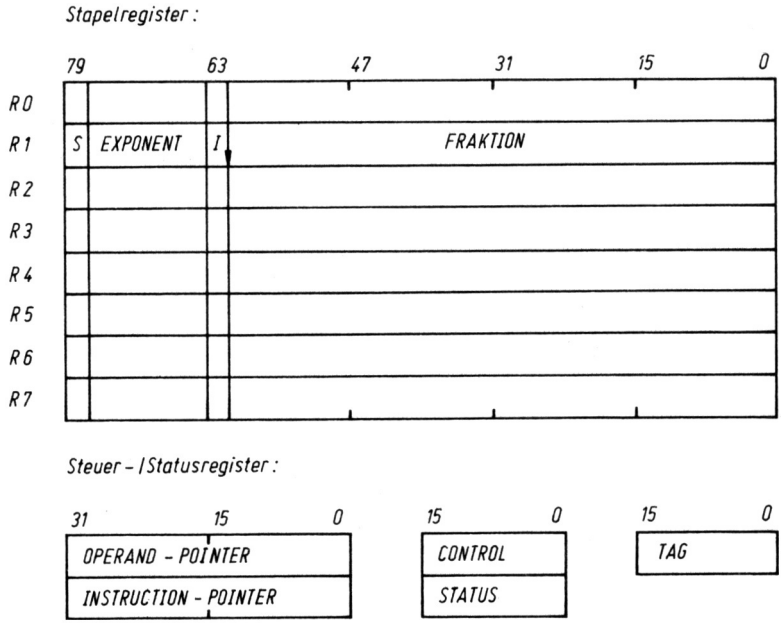


Bild 7.4. Registersatz des K1810WM87

7.1.6. Busoperationen

Das Businterface des K1810WM87 muß folgende Aufgaben erfüllen:

- Dekodieren von Befehlskodes,
- Laden von Speicheroperanden,
- Abspeichern von Ergebnissen.

Die Befehle für den K1810WM87 werden vom K1810WM86 eingelesen. Der K1810WM87 beobachtet alle Busleitungen, um Kodelesezyklen zu erkennen, und dekodiert parallel alle Befehle mit. Beginnt ein Befehl mit ESC (11011xxxB), so werden die Adresse und der Kode im Befehlszeiger abgelegt, und der Befehl wird ausgeführt. Der K1810WM86 liest inzwischen weitere Befehle ein und führt sie aus. Der Coprozessor dekodiert auch diese, ignoriert sie aber, bis ein neuer Escape-Kode erkannt wird.
ESC-Befehle können je nach Adreßmodusbyte Register- oder Speicheroperanden adressieren. Bei Registeroperanden werden keine Buszyklen generiert. Ist ein Speicheroperand angegeben, so berechnet der K1810WM86 die effektive Adresse und führt einen Lesezyklus aus, dessen Daten er ignoriert ("Dummy Read"). Dabei ist es belanglos, ob die Ausführung des Befehls durch den Coprozessor ein Datenlesen oder -schreiben bewirken soll. Die weiteren Operationen sind nun Sache des K1810WM87 und laufen wie folgt ab:

Speicheroperand lesen
Der Coprozessor speichert die vom K1810WM86 berechnete und ausgegebene Speicheradresse im Operandenzeiger ab und übernimmt auch das auf dem Bus erscheinende Datenwort. Ist der Operand als Wort vereinbart, sind keine Buszyklen mehr erforderlich. In der Regel sind aber Operanden für den K1810WM87 zwei, vier oder fünf Worte lang. In diesem Fall fordert der Coprozessor über die /RQ/GT-Leitung den Bus an, erhöht automatisch die Operandenadresse um jeweils 2 und generiert selbst die restlichen Speicherzyklen, um den Operanden vollständig einzulesen.
Falls die Operandenadresse ungerade ist, liest der K1810WM86 das erste Wort byteweise. Der Coprozessor liest dann (falls der Operand entsprechend lang ist) ein weiteres Byte und den Rest des Operanden wortweise. Es wird jedoch empfohlen, Variable für den K1810WM87 auf geraden Adressen zu vereinbaren. (Dazu kann die EVEN-Direktive benutzt werden.)

Speicheroperand schreiben
Soll der K1810WM87 einen Wert in den Speicher übertragen, so speichert er die vom K1810WM86 berechnete und ausgegebene Speicheradresse ebenfalls ab, ignoriert aber die auf dem Bus erscheinenden Daten. (Der vom K1810WM86 generierte Zyklus ist ja ein Lesezyklus.) Hat er die Operation soweit ausgeführt, daß er den Operanden ausgeben kann, fordert er über die /RQ/GT-Leitung den Bus an und generiert mit Hilfe der "gemerkten" Adresse so viele Speicherschreibzyklen, wie zum Abspeichern des Operanden notwendig sind.

In den Speicherzyklen, in denen der K1810WM87 selbst aktiv ist, steuert er den Bus, indem er die Signale /S0 bis /S2 analog zum K1810WM86 aktiviert. (Die Speicheroperationen des K1810WM86 sind im 2. Abschnitt ausführlich beschrieben.)

7.2. Programmierung des K1810WM87

7.2.1. Adressierungsarten

Operanden für K1810WM87-Befehle können sein:

- Speicherdaten,
- Stapelregister,
- Status- und Steuerregister.

Die Speicheradressierung übernimmt der K1810WM86, der die effektive Adresse berechnet. Deshalb kann auf Speicherdaten mit allen Adressierungsarten des K1810WM86 zugegriffen werden.
Die Status- und Steuerregister können nur implizit durch spezielle Befehle angesprochen werden.

Stapelregister können entweder implizit oder explizit adressiert werden. Die implizite Form wird von Befehlen angewendet, die mit der Stapelspitze operieren (z.B. FSQRT). Explizit werden Stapelregister in der Form ST(i) , i=0,...7 adressiert. Der Index i zeigt auf das i-te Register nach der Stapelspitze. Die Angaben ST und ST(0) sind identisch.
Die Stapelregister werden also nicht physisch adressiert, sondern relativ zur aktuellen Stapelspitze.

Enthält beispielsweise der Stapelzeiger den Wert 2, so addiert der Befehl FADD ST,ST(1) die Inhalte der physischen Register 2 und 3. Wird nun mit einem Befehl FLD ein Wert geladen, so wird der Stapelzeiger auf 1 gesetzt, und der Befehl FADD ST,ST(1) würde jetzt die Inhalte der Register 1 und 2 addieren.

Dieses Verfahren erleichtert die Arbeit mit Unterprogrammen. Ein Anwenderprogramm kann eine K1810WM87-Prozedur aufrufen, ohne wissen zu müssen, welche Register momentan frei sind oder wo die Prozedur ihre Parameter erwartet. Es lädt einfach den Stapel, und die Prozedur adressiert den Stapel relativ über ST(i). Ein anderes Programm kann die gleiche Prozedur aufrufen, aber mit anderen physischen Registern arbeiten.

7.2.2. Spezielle Werte

Normalerweise arbeitet der K1810WM87 mit Operanden, die intern im normierten Temporary-Real-Format dargestellt werden und reelle Zahlen repräsentieren. Unter bestimmten Bedingungen können jedoch spezielle Werte als Operand oder Ergebnis auftreten, die hier beschrieben werden sollen. Dazu gehören die Werte Null und Unend-

lich, nichtnormierte Zahlen und besondere Kodierungen, die nicht als Zahlen interpretiert werden. (Diese Werte werden nur selten auftreten und für viele Anwendungen ohne Bedeutung sein.)

Null:
Der Wert 0 ist ein normaler Operand, wird aber im TAG-Wort angezeigt. Im BCD- und Real-Format kann der Wert 0 positiv oder negativ sein.
Neben der wirklichen Null, deren Signifikand und Exponent 0 sind, können auch Pseudo-Nullen auftreten. (Etwa bei der Multiplikation zweier nichtnormierter Zahlen, wenn mehr als 64 führende Nullen entstehen.) Sie haben einen von 0 verschiedenen Exponenten und werden wie unnormale Werte behandelt, mit Ausnahme der Vergleichsbefehle, wo sie als wirkliche Null interpretiert werden.

Denormierte Zahlen (Denormals):
Der kleinste darstellbare Wert ist normalerweise durch den Bias des Exponenten bestimmt.
Ist ein Operand so klein, daß er im vereinbarten Format nicht dargestellt werden kann (der Exponent ist kleiner als der Bias), wird die Underflow-Exception gesetzt. Ist diese unmaskiert, wird ein Interrupt gesendet und die Operation abgebrochen. Bei Registeroperanden wird zum Exponenten 24576 addiert, um die Zahl in den gültigen Bereich zu transformieren. (Die Exception-Prozedur kann diesen Wert wieder subtrahieren.) Ist die Underflow-Exception maskiert, wird die Berechnung fortgesetzt. In diesem Fall wird die Zahl denormiert. Der Exponent wird auf 0 gesetzt, ebenfalls das Integerbit, und bei Bedarf werden weitere führende Nullen eingefügt, um die Zahl korrekt darzustellen. (Erst 64 führende Nullen produzieren als Ergebnis Null.)
Solche denormierten Zahlen werden im TAG-Wort als spezieller Wert gekennzeichnet. Sie führen zu einem Verlust an Genauigkeit (die niederwertigsten signifikanten Bits gehen verloren), aber es wird ein abrupter Underflow-Abbruch vermieden.

Unnormale Zahlen (Unnormals):
Ähnlich wie denormierte sind unnormale Zahlen durch führende Nullen im Signifikanden gekennzeichnet. Ihr Exponent kann aber beliebig sein. Sie können auftreten, wenn sie als Quelloperand so initialisiert sind, oder im Ergebnis von Operationen mit denormierten Zahlen. Sie werden als gültige Operanden behandelt.

Unendlich:
Unendlich kann als Quelloperand initialisiert sein oder als Ergebnis in folgenden Fällen entstehen:

- Eine Division durch Null liefert Unendlich, wenn die Zero-Divide-Exception maskiert ist.
- Falls das Ergebnis einer Operation zu groß für das Format des Zieloperanden ist, wird Unendlich geliefert, wenn die Overflow-Exception maskiert ist. Ist sie nicht maskiert und das Ergebnis steht in einem Stapelregister, so wird 24576 vom Exponenten subtrahiert und ein Interrupt gesendet.

Der Wert Unendlich kann positiv oder negativ sein. Er wird stets durch einen Exponenten 11...11 und einen Signifikanden 1,00...00 dargestellt. Im TAG-Wort wird Unendlich als spezieller Wert gekennzeichnet.

Indefinit-Kode:
Dies ist ein spezieller Wert, der im Ergebnis ungültiger Operationen entsteht, wenn die Invalid-Operation-Exception maskiert ist.
Im Real-Format wird Indefinit durch einen speziellen NAN-Kode dargestellt: S = 1, Exponent = 11..11, Signifikand = 1,100..00.
Im BCD-Format wird das höchstwertige Wort auf FFFFH gesetzt. (Normalerweise sind die Bits 72 - 78 Null.)
Im Integer-Format existiert keine Darstellung für Indefinit, deshalb wird an dessen Stelle der kleinste mögliche Wert ausgegeben (S = 1, alle anderen Bits = 0).

NAN (Not A Number):
NANs sind spezielle Werte, die durch einen Exponenten 11...11 charakterisiert sind. Das Vorzeichen und der Signifikand können beliebig sein (außer 1,00...00 , dem Kode für Unendlich). Die Anwendung von NANs als Operanden führt stets zur Invalid-Operation-Exception, mit Ausnahme der transzendenten Funktionen, die ihre Operanden nicht prüfen.
Bei der Anwendung von NANs ist dem Programmierer freier Spielraum gelassen. Insbesondere für Test- und Diagnosezwecke sind sie gut geeignet. Ein Beispiel sind Variable, die mit verschiedenen NAN-Kodes initialisiert werden, bevor ein Anwenderprogramm ihnen normale Werte zuweist. Greift eine K1810WM87-Prozedur darauf zu, ohne daß die notwendigen Werte eingetragen wurden, kann eine Exception-Prozedur zur Fehlerbehandlung gestartet werden.

7.2.3. Befehlssynchronisation mit WAIT

Der K1810WM87 dekodiert alle Befehle parallel zum K1810WM86, wenn sie von diesem eingelesen werden. Der Host-Prozessor benötigt aber zur "Ausführung" eines ESC-Befehls nur zwei Takte, während der Coprozessor für die Ausführung des gleichen Befehls stets mehr Zeit benötigt. Inzwischen hat der K1810WM86 längst neue Befehle eingelesen und dekodiert. Auch der Coprozessor dekodiert diese parallel zur Ausführung der laufenden Operation. So lange es keine ESC-Befehle sind, tritt kein Problem auf - beide Prozessoren arbeiten dann parallel.
Befehle werden also synchron dekodiert, aber asynchron ausgeführt. In zwei Situationen ist dennoch eine Synchronisierung der Befehlsausführung nötig:

- Der K1810WM87 darf keinen ESC-Befehl dekodieren, wenn eine laufende Operation noch nicht beendet ist (das heißt, wenn BUSY aktiv ist).
  Eine Ausnahme bilden die "NO-WAIT"-Steuerbefehle, die der K1810WM87 parallel zu einem anderen Befehl ausführen kann.

- Der K1810WM86 darf nicht auf einen Speicheroperanden zugreifen, den er als Ergebnis einer K1810WM87-Operation erwartet, bevor dieser das Ergebnis wirklich abgespeichert hat. Andernfalls sind Programmfehler die Folge.

Diese Situation wird mit Hilfe des WAIT-Befehls gelöst, der den K1810WM86 so lange an der Ausführung eines neuen Befehls hindert, bis das BUSY-Signal inaktiv wird.
Der erste kritische Fall kann ausgeschlossen werden, wenn jedem ESC-Befehl ein WAIT-Befehl vorangestellt wird. Diese Aufgabe übernimmt der Assembler, der vor jedem K1810WM87-Maschinenbefehl automatisch einen WAIT-Kode generiert. Der Programmierer muß diese Situation also nicht berücksichtigen (der WAIT erscheint nicht im Quellkode).
Die zweite kritische Situation muß aber vom Programmierer ausgeschaltet werden. Er muß vor jedem Befehl, der auf ein vom Coprozessor abzuspeicherndes Ergebnis zugreift, explizit einen WAIT-Befehl programmieren. Dazu sollte die Mnemonik FWAIT benutzt werden. Der Assembler unterscheidet drei Formen des WAIT-Befehls:

- ein implizites WAIT vor ESC-Kodes,
- eine explizite FWAIT-Mnemonik,
- eine explizite WAIT-Mnemonik.

Alle Formen generieren ein und denselben Maschinenkode. Die ersten beiden Formen werden aber eliminiert, wenn das Programm mit einem Software-Emulator arbeitet. Ansonsten kann die Gefahr eines endlosen WAIT-Zustandes eintreten, wenn kein Coprozessor den /TEST-Eingang des K1810WM86 schaltet.

7.2.4. Exception-Behandlung

Bei den meisten Operationen prüft der K1810WM87 die Operanden und Ergebnisse auf Ausnahme- oder Fehlerbedingungen (Exceptions). Drei solche Bedingungen werden bereits vor der Ausführung einer Operation geprüft:

Ungültige Operation (Invalid Operation)
Sie kann unter folgenden Bedingungen auftreten:

- Laden eines Stapelregisters, wenn keines mehr frei ist,
- Speichern aus dem Stapel, wenn alle Register frei sind,
- Zugriff auf einen NAN-Operanden,
- Verbotene arithmetische Operation (z.B. 0:0, Wurzel aus negativen Zahlen, Unendlich - Unendlich),
- Division durch eine denormierte Zahl.

Division durch Null (Zero Divide)
Sie tritt auf, wenn ein endlicher, von Null verschiedener Operand durch Null dividiert wird.

Denormaler Operand (Denormal Operand)
Wenn eine denormierte Zahl als Quelloperand entdeckt wird, tritt diese Exception auf.

In diesen Fällen wird die Operation nicht ausgeführt, bevor die Exception behandelt wurde.
Drei weitere Bedingungen werden erst am Ende von Operationen geprüft:

Überlauf (Overflow)
Ein Überlauf tritt auf, wenn der Exponent des Ergebnisses zu groß ist, um dieses im Format des Zieloperanden darzustellen.

Unterschreitung (Underflow)
Eine Unterschreitung tritt auf, wenn der Exponent des Ergebnisses zu klein ist, um dieses im Format des Zieloperanden darzustellen.

Genauigkeit (Precision)
Diese Exception signalisiert, daß das Ergebnis nicht exakt im Format des Zieloperanden dargestellt werden kann und gerundet wird.

Das Auftreten einer Exception signalisiert der K1810WM87, indem er das zugehörige Exception-Flag im Statusregister auf 1 setzt. Seine weitere Reaktion ist nun abhängig vom Steuerwort. Dieses enthält für jede Exception ein Maskenbit.
Ist das Maskenbit auf 1 gesetzt (maskierte Exception), so führt der K1810WM87 seine implementierte "maskierte Exception-Antwort" aus. Das bedeutet, daß der Coprozessor über einen eigenen Exception-Handler verfügt, der automatisch reagiert. Die Operation wird anschließend fortgesetzt. Damit wird eine für die meisten Anwendungsfälle zumutbare Reaktion ausgeführt, ohne die Berechnung zu unterbrechen (Tafel 7.3).
Für besondere Anwendungsfälle können Exceptions unmaskiert bleiben, indem das Maskenbit auf 0 gesetzt wird. In diesen Fällen generiert der K1810WM87 einen Interrupt, um das Anwenderprogramm zur Behandlung der Exception zu veranlassen. Der Programmierer muß also eine Interruptprozedur vorbereiten, die seinen speziellen Exception-Handler startet.
Alle Exception-Flags sind statisch; das heißt, sie bleiben so lange gesetzt, bis sie vom Programm zurückgesetzt werden. Dazu kann der Befehl FCLEX angewendet werden, oder der Coprozessor wird neu initialisiert. Sind alle Exceptions maskiert, kann eine komplette Prozedur ohne Unterbrechung zu Ende geführt werden, um zuletzt auf eventuell aufgetretene Exceptions zu prüfen.
Es soll noch angemerkt werden, daß Exceptions selten auftreten. Die Invalid-Operation-Exception signalisiert meist einen Programmfehler. Die am häufigsten auftretende Precision-Exception kann meist ignoriert werden.

Tafel 7.3. Exception-Behandlung

Typ	Behandlung	
	maskiert	unmaskiert
I	Ist ein Operand NAN, wird er als Ergebnis eingesetzt; ist kein Operand NAN, wird Indefinit eingesetzt.	Interrupt.
Z	Es wird Unendlich als Ergebnis eingesetzt.	Interrupt.
D	Die Operation wird ausgeführt.	Interrupt.
O	Es wird Unendlich als Ergebnis eingesetzt.	Bei Registeroperanden wird 24576 vom Exponenten subtrahiert, Interrupt.
U	Das Ergebnis wird denormalisiert.	Bei Registeroperanden wird 24576 zum Exponenten addiert, Interrupt.
P	Das Ergebnis wird gerundet.	Das Ergebnis wird gerundet, Interrupt.

7.2.5. Interrupts

Der K1810WM87 kann über seinen INT-Ausgang Interruptanforderungen an den K1810WM86 - seinen Host-Prozessor - senden. Ursache ist stets eine unmaskierte Exception. Ist die Exception maskiert, führt der K1810WM87 seine interne Exception-Behandlung durch.
Bei nichtmaskierten Exceptions wird das Interruptanforderungsflag IR im Statuswort gesetzt und die Interruptmaske im Steuerwort getestet. Sind Interrupts erlaubt (ID=0), aktiviert der Coprozessor sein INT-Signal und wartet auf die Exception-Behandlung durch das Programm.
Falls Interrupts verboten sind (ID=1), überprüft der K1810WM87, ob der Host-Prozessor im WAIT-Zustand ist. (Da alle Befehle parallel dekodiert werden, weiß er das.) Hat der K1810WM86 keinen WAIT-Befehl abgearbeitet, werden die folgenden Befehle dekodiert und der nächste ESC-Befehl ausgeführt. Normalerweise sollte das der Befehl FNENI sein, der die Interruptmaske zurücksetzt. (Es können nur "NO-WAIT"-Kodes ausgeführt werden, andernfalls wird vorher ein WAIT-Befehl generiert.) Durch andere Befehle (z.B. FNCLEX) kann der Interrupt zerstört werden.
Befindet sich der K1810WM86 im WAIT-Zustand, während im K1810WM87 ein unbearbeiteter Interrupt ansteht, besteht die Gefahr eines endlosen WAIT-Zustands. Der Host-Prozessor wartet darauf, daß die BUSY-Leitung inaktiv wird, während der Coprozessor diese aktiv hält und darauf wartet, daß die Exception behandelt wird. Diese kritische Situation umgeht der Coprozessor, indem er trotz ge-

setzter Interruptmaske die INT-Leitung aktiviert. Dadurch besteht die Möglichkeit, daß die Exception behandelt wird und ein normaler Zustand hergestellt wird, denn der K1810WM86 bearbeitet Interruptanforderungen auch im WAIT-Zustand.
Die Gefahr eines endlosen WAIT-Zustands bleibt aber dennoch bestehen, denn der Interrupt des K1810WM87 kann auch von einer Anerkennung ausgeschlossen sein, wenn der entsprechende Eingang am Interrupt-Controller des Host-Prozessors maskiert ist oder Interrupts durch einen CLI-Befehl generell verboten sind. In diesem Fall könnte nur noch ein anderer, zugelassener Interrupt (z.B. ein NMI) den Interruptpfad für den Coprozessor freimachen. Schließlich kann auch ein endloser WAIT-Zustand provoziert werden, wenn der Interruptpfad zwar frei ist und die Exception-Prozedur startet, diese aber selbst einen WAIT-Befehl ausführt, bevor die Exception und damit der Interrupt zurückgesetzt wurde. Deshalb ist bei der Programmierung von Exception-Handlern höchste Aufmerksamkeit geboten. Prinzipiell sollte kein ESC-Befehl abgearbeitet werden, dem ein WAIT vorangeht, wenn Interrupts verboten sind und die Möglichkeit besteht, daß der K1810WM87 einen Interrupt anfordern könnte. Außerdem muß die Interruptprozedur die Exception zurücksetzen, bevor in das unterbrochene Programm zurückgekehrt wird. Ansonsten wird sofort ein neuer Interrupt generiert und das Programm gerät in eine endlose Schleife.

7.2.6. Initialisierung

Der K1810WM87 kann durch ein RESET-Signal oder vom Programm initialisiert werden. Dazu können die Befehle FINIT oder FSAVE angewendet werden. In allen Fällen nimmt der K1810WM87 den in Tafel 7.4 beschriebenen Zustand ein.

Tafel 7.4. Zustand des K1810WM87 nach der Initialisierung

Registerfeld	Wert	Bedeutung
Steuerwort:		
Unendlich	IC = 0	Projektives Modell
Rundungsverfahren	RC = 00	Runde zum nächstliegenden Wert
Genauigkeit	PC = 11	64 signifikante Bits
Interruptmaske	ID = 1	Interrupts verboten
Exception-Masken	XM = 1	Exceptions maskiert
Statuswort:		
BUSY-Flag	B = 0	nicht besetzt
Ergebniskode	C = ?	undefiniert
Stapelzeiger	TOS = 000	Stapel leer
Interruptflag	IR = 0	kein Interrupt
Exception-Flags	X = 0	keine Exceptions
TAG-Wort	T(i) = 11	alle Register frei

X = P, U, O, Z, D, I

Die Initialisierung mit FSAVE bewirkt außerdem, daß der vorhergehende Zustand im Speicher konserviert wird. Das hat den Vorteil, daß auch Unterprogramme den Coprozessor initialisieren können. Bei der Rückkehr kann mit dem Befehl FRSTOR der alte Zustand wieder rekonstruiert werden.

7.3. Befehlssatz des K1810WM87

Der Befehlssatz des K1810WM87 umfaßt 69 Operationen. Diese werden mit Hilfe des ESC-Kodes gebildet und im folgenden beschrieben. Viele Befehle operieren nur mit Stapelregistern als Operanden. Datentransfer-, Arithmetik- und Vergleichsbefehle können auch auf Operanden im Speicher zugreifen. Bild 7.5 zeigt die Formate der Speicherdaten. Einige der Steuerbefehle operieren auch mit Speicheroperanden, deren Formate den Steuer- und Statusregistern entsprechen.

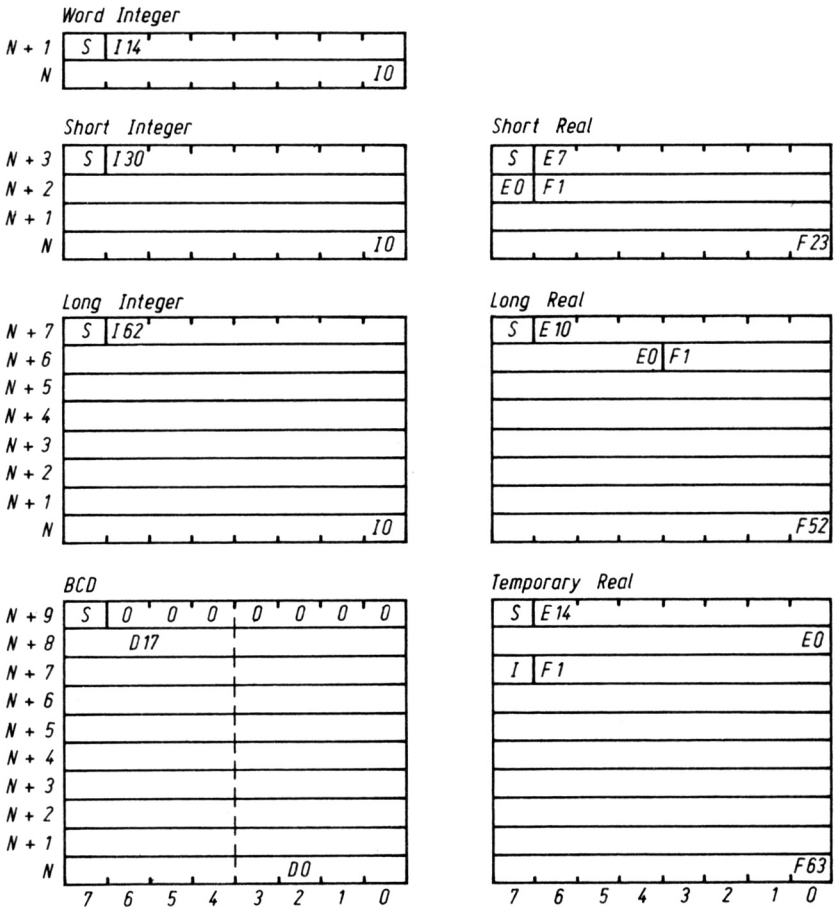


Bild 7.5. Speicherformate der K1810WM87-Operanden

Für die Benutzung der Operandentypen durch die K1810WM87-Befehle gelten folgende Regeln:

1. Die Stapelregister können von allen Befehlen benutzt werden, die mit Real-Zahlen operieren. Manche Befehle erfordern implizit die Anwendung der Stapelspitze ST bzw. ST und ST(i).
2. Speicherdaten im Short- und Long-Real-Format können von Datentransfer-, Arithmetik- und Vergleichsbefehlen benutzt werden, die mit Real-Zahlen operieren.
3. Speicherdaten im Temporary-Real-Format können nur von den Datentransferbefehlen FLD und FSTP angewendet werden.
4. Speicherdaten im Integer-Format können von allen Datentransfer-, Arithmetik- und Vergleichsbefehlen benutzt werden, die mit Integerdaten operieren.
5. Speicherdaten im BCD-Format können nur die Datentransferbefehle FBLD und FBSTP anwenden.

Im Anhang B ist eine vollständige Befehlsliste mit Maschinenkodes, Ausführungszeiten und Exceptions enthalten.

7.3.1. Datentransferbefehle

Diese Befehle transportieren Operanden innerhalb des Registerstapels und zwischen Stapelspitze und Speicher. Bei jedem Befehl wird das TAG-Wort aktualisiert.

FLD src (Load Real)
Der Befehl FLD erniedrigt den Stapelzeiger um 1 und lädt die Stapelspitze mit einem Operanden im Real-Format. Quelle kann eines der anderen Stapelregister oder der Speicher sein. Speicheroperanden werden in Temporary-Real-Format konvertiert.

FILD src (Load Integer)
Der Befehl FILD erniedrigt den Stapelzeiger um 1 und lädt die Stapelspitze mit einem Speicheroperanden im Integer-Format. Dieser wird dabei automatisch in Temporary-Real-Format konvertiert.

FBLD src (Load BCD)
Der Befehl FBLD erniedrigt den Stapelzeiger um 1 und lädt die Stapelspitze mit einem Speicheroperanden im Dezimal-Format. Dieser wird dabei automatisch in Temporary-Real-Format konvertiert. Das Vorzeichen bleibt erhalten, es treten keine Rundungsfehler auf. Der Befehl testet nicht auf fehlerhafte Digits (AH bis FH)!

FST dest (Store Real)
Der Befehl FST überträgt den Inhalt der Stapelspitze in den Speicher oder in ein anderes Stapelregister. Speicheroperanden können vom Typ Short oder Long Real sein und werden in das entsprechende Format konvertiert. Dabei wird gemäß der Rundungsvorschrift im Steuerwort gerundet.

FSTP dest (Store Real And Pop)
Der Befehl FSTP führt die gleiche Operation wie FST aus; anschließend wird aber die Stapelspitze gelöscht und der Stapelzeiger durch Erhöhen auf das nächste Register gesetzt. Dieser Befehl ermöglicht auch den Transfer zu einem Speicheroperanden im Temporary-Real-Format.

FIST dest (Store Integer)
Der Inhalt der Stapelspitze wird in das Format Word oder Short Integer konvertiert, gemäß der Rundungsvorschrift im Steuerwort gerundet und in den Speicher übertragen.

FISTP dest (Store Integer And Pop)
Der Befehl FISTP bewirkt die gleiche Operation wie FIST; anschließend wird aber das Register gelöscht und der Stapelzeiger um 1 erhöht. Dieser Befehl gestattet auch das Übertragen von Daten im Long-Integer-Format.

FBSTP dest (Store BCD And Pop)
Der Inhalt der Stapelspitze wird in eine gepackte BCD-Zahl konvertiert und in den Speicher übertragen. Anschließend wird der Inhalt der Stapelspitze gelöscht und der Stapelzeiger um 1 erhöht. Es wird stets ein ganzzahliges Ergebnis geliefert, indem 0,5 addiert und der nichtganzzahlige Anteil abgeschnitten wird.

FXCH (dest) (Exchange)
Der Inhalt des Zieloperanden (eines der Stapelregister) wird mit dem Inhalt der Stapelspitze vertauscht. Ist kein Operand angegeben, wird automatisch ST(1) angenommen.
Da viele Befehle nur mit der Stapelspitze arbeiten, bietet der FXCH-Befehl eine einfache Möglichkeit, auch andere Register solchen Befehlen zugänglich zu machen.

7.3.2. Arithmetikbefehle

Der K1810WM87 bietet Befehle zur Addition, Subtraktion, Multiplikation und Division. Darüber hinaus stehen Befehle für spezielle Funktionen wie Quadratwurzel, Absolutwert u.ä. zur Verfügung. Für Subtraktion und Division existieren symmetrische Operationen, so daß sechs Grundoperationen möglich sind:

```
ADD  : dest <= dest + src
SUB  : dest <= dest - src
SUBR : dest <= src  - dest
MUL  : dest <= dest x src
DIV  : dest <= dest : src
DIVR : dest <= src  : dest
```

Als Operanden können bei diesen Operationen die Stapelspitze, andere Stapelregister oder Speicherdaten benutzt werden. Einige Befehlsformen verwenden implizite Operanden (Tafel 7.5).

Speicheroperanden werden stets in Temporary-Real-Format konvertiert.
Die meisten Befehle lassen den Stapelzeiger unbeeinflußt; nur die POP-Form (FXXXP) löscht im Anschluß an die Operation die Stapelspitze und erhöht den Stapelzeiger um 1.

Tafel 7.5. Operanden der Arithmetikoperationen

mnemonische Form	Operanden (dest,src)
FXXX	ST(1) , ST
FXXX ST(i),ST	ST(i) , ST
FXXX ST,ST(i)	ST , ST(i)
FXXXP ST(i),ST	ST(i) , ST
FXXX short real	ST , mem32
FXXX long real	ST , mem64
FIXXX word integer	ST , mem16
FIXXX short integer	ST , mem32

XXX kann ADD, SUB, SUBR, MUL, DIV, DIVR sein

FADD (dest,src) (Add Real)
FADDP dest,src (Add Real And Pop)
FIADD src (Add Integer)
Diese Befehle addieren den Quelloperanden zum Zieloperanden, der durch die Summe ersetzt wird. Der Befehl FADDP inkrementiert den Stapelzeiger nach der Operation.
Der Befehl FADD ST,ST(0) verdoppelt den Inhalt der Stapelspitze.

FSUB (dest,src) (Subtract Real)
FSUBP dest,src (Subtract Real And Pop)
FISUB src (Subtract Integer)
Diese Befehle subtrahieren den Quelloperanden vom Zieloperanden, der durch die Differenz ersetzt wird. Der Befehl FSUBP inkrementiert den Stapelzeiger nach der Operation.

FSUBR (dest,src) (Subtract Reverse Real)
FSUBRP dest,src (Subtract Reverse Real And Pop)
FISUBR src (Subtract Reverse Integer)
Diese Befehle subtrahieren den Zieloperanden vom Quelloperanden. Der Zieloperand wird durch die Differenz ersetzt. Der Befehl FSUBRP inkrementiert den Stapelzeiger nach der Operation.

FMUL (dest,src) (Multiply Real)
FMULP dest,src (Multiply Real And Pop)
FIMUL src (Multiply Integer)
Diese Befehle multiplizieren den Zieloperanden mit dem Quelloperanden. Der Zieloperand wird durch das Produkt ersetzt. Der Befehl FMULP inkrementiert den Stapelzeiger nach der Operation.
Der Befehl FMUL ST,ST(0) quadriert die Stapelspitze.

FDIV (dest,src) (Divide Real)
FDIVP dest,src (Divide Real And Pop)
FIDIV src (Divide Integer)
Diese Befehle dividieren den Zieloperanden durch den Quelloperanden. Der Zieloperand wird durch den Quotienten ersetzt. Der Befehl FDIVP inkrementiert den Stapelzeiger nach der Operation.

FDIVR (dest,src) (Divide Reverse Real)
FDIVRP dest,src (Divide Reverse Real And Pop)
FIDIVR src (Divide Reverse Integer)
Diese Befehle dividieren den Quelloperanden durch den Zieloperanden, der durch den Quotienten ersetzt wird. Der Befehl FDIVRP inkrementiert den Stapelzeiger nach der Operation.

FSQRT (Square Root)
Dieser Befehl ersetzt den Inhalt der Stapelspitze durch seine Quadratwurzel. (Die Wurzel aus -0 ist als -0 definiert, ansonsten führen Wurzeln aus negativen Werten zur Invalid-Operation-Exception.)

FSCALE (Scale)
Dieser Befehl führt folgende Operation aus:

$$ST <= ST \times 2^{ST(1)}$$

Dazu wird der Inhalt in ST(1) als ganze Zahl interpretiert, die zum Exponentfeld von ST addiert wird. Die Zahl in ST(1) muß im Bereich -2^{15} bis $+2^{15}$ liegen, sonst wird ein fehlerhaftes Ergebnis produziert, ohne eine Exception zu setzen. Deshalb sollte ST(1) mit einer Word-Integer-Zahl geladen werden.
Die SCALE-Funktion ermöglicht die schnelle Multiplikation oder Division mit Zweierpotenzen.

FPREM (Partial Remainder)
Dieser Befehl führt folgende Operation aus:

ST <= ST modulo ST(1)

(Modulo-Division der Stapelspitze mit ST(1); das heißt, die Zahl in ST wird so oft durch die Zahl in ST(1) dividiert, bis ein Rest entsteht, der kleiner als der Modul ist.) Dabei wird ein exaktes Ergebnis produziert; das Vorzeichen bleibt erhalten.
Der Befehl kann Größenunterschiede bis 2^{64} verarbeiten, indem eine sukzessive Subtraktion durchgeführt wird. Wenn die Operation beendet ist, wird das C2-Bit im Statuswort auf 0 gesetzt. Übersteigt der Größenunterschied diesen Wert, so liefert die Operation nur einen partiellen Rest (größer als der Modul), und das C2-Bit wird auf 1 gesetzt. Dann muß der Befehl wiederholt werden. Um dies zu entscheiden, kann das C2-Bit getestet oder der Inhalt von ST und ST(1) verglichen werden.
Dieser Befehl kann angewendet werden, um das Argument periodischer Funktionen in den gültigen Bereich zu transformieren.

FRNDINT (Round To Integer)
Dieser Befehl rundet den Inhalt der Stapelspitze zu einer ganzen Zahl gemäß der Rundungsvorschrift im Steuerwort.

FXTRACT (Extract)
Dieser Befehl zerlegt den Inhalt der Stapelspitze in seinen Signifikanden und den wahren Exponenten. Dabei wird der Stapelzeiger erhöht und der Signifikand als Real-Zahl in die neue Stapelspitze transportiert, während der Exponent nach Subtraktion des Bias ebenfalls als Real-Zahl in ST(1) abgespeichert wird. Nach der Operation enthält also ST eine Zahl, deren Signifikand dem des Operanden entspricht und deren Exponent 3FFFH ist, was dem wahren Exponenten 0 entspricht. ST(1) enthält den Wert des wahren Exponenten des Operanden.
Hat der Operand den Wert ±0, so werden beide Register auf ±0 gesetzt.
Dieser Befehl ist nützlich, wenn Zahlen im Temporary-Real-Format für Dezimalanzeigen aufbereitet werden müssen.

FABS (Absolute)
Der Befehl FABS bildet den Absolutwert des Inhalts der Stapelspitze, das heißt, das Vorzeichenbit wird auf 0 gesetzt.

FCHS (Change Sign)
Der Befehl FCHS invertiert das Vorzeichen des Inhalts der Stapelspitze.

7.3.3. Vergleichsbefehle

Diese Befehle vergleichen den Inhalt der Stapelspitze mit einem Quelloperanden, der ein anderes Stapelregister oder eine Speichervariable sein kann. Das Ergebnis wird durch die Bits C0 bis C3 des Statuswortes reflektiert. Dieses kann in den Speicher geladen und dort vom K1810WM86 getestet werden. Tafel 7.6 zeigt die Kodierungen der Ergebnisbits für die meisten Befehle dieser Gruppe.

Tafel 7.6. Kodierung der Vergleichsergebnisse

C3	C0	COMpare-Befehle	TST-Befehl
0	0	ST > src	ST > 0
0	1	ST < src	ST < 0
1	0	ST = src	ST = 0
1	1	ST ? src	ST ?

? bedeutet, daß ein Operand Unendlich oder NAN ist

FCOM (src) (Compare Real)
FCOMP (src) (Compare Real And Pop)
Diese Befehle vergleichen eine Real-Zahl mit dem Inhalt der Stapelspitze. Ist kein Operand angegeben, wird ST(1) angenommen. Das Ergebnis wird nach Tafel 7.6 kodiert.
Der Befehl FCOMP löscht anschließend den Inhalt der Stapelspitze und erhöht den Stapelzeiger um 1.

FCOMPP (Compare Real And Pop Twice)
Dieser Befehl vergleicht die Inhalte der Register ST und ST(1), löscht danach beide Register und erhöht den Stapelzeiger um 2. Das Ergebnis wird nach Tafel 7.6 kodiert.

FICOM src (Compare Integer)
FICOMP src (Compare Integer And Pop)
Diese Befehle vergleichen eine Integer-Zahl aus dem Speicher mit dem Inhalt der Stapelspitze. Das Ergebnis wird nach Tafel 7.6 kodiert.
Der Befehl FICOMP löscht anschließend den Inhalt der Stapelspitze und erhöht den Stapelzeiger um 1.

FTST (Test Against Zero)
Der Befehl FTST vergleicht den Inhalt der Stapelspitze mit dem Wert 0. Das Ergebnis wird nach Tafel 7.6 kodiert.

FXAM (Examine)
Der Befehl FXAM überprüft den Inhalt der Stapelspitze hinsichtlich spezieller Werte. Das Ergebnis der Operation ist nach Tafel 7.7 kodiert und kann durch Testen des Statuswortes ermittelt werden.

Tafel 7.7. Ergebnis des FXAM-Befehls

Bit C3	C2	C0	Ergebnis	Bit C1	Ergebnis
0	0	0	Unnormal	0	+
0	0	1	NAN	1	-
0	1	0	Normal		
0	1	1	Unendlich		
1	0	0	Null	(Bit C1 repräsentiert das Vorzeichen)	
1	1	0	Denormal		
1	X	1	Frei		

7.3.4. Transzendente Funktionen

Die Befehle dieser Gruppe erlauben die Berechnung von trigonometrischen, hyperbolischen, logarithmischen und exponentiellen Funktionen. Die Befehle setzen voraus, daß die Operanden gültig sind und innerhalb des jeweils definierten zulässigen Bereichs liegen. Spezielle Werte als Operanden liefern fehlerhafte Ergebnisse, ohne Exceptions zu setzen.

Der Befehl FPREM kann benutzt werden, um die Argumente von periodischen Funktionen in den zulässigen Bereich zu transformieren.

FPTAN (Partial Tangent)
Dieser Befehl berechnet die Funktion

$$y / x = \tan w$$

Der Winkel w wird aus der Stapelspitze genommen und muß im Bereich $0 < w < \pi/4$ liegen. Das Ergebnis wird als Verhältnis in zwei Stapelregistern abgelegt. y ersetzt w, dann wird der Stapelzeiger dekrementiert und x in der neuen Stapelspitze abgelegt, so daß nach der Operation x in ST und y in ST(1) steht.

FPATAN (Partial Arctangent)
Dieser Befehl berechnet die Funktion

$$w = \arctan (y/x)$$

Der Wert y wird aus ST(1), x aus ST genommen. Die Operanden müssen die Beziehung $0 < y < x < \infty$ einhalten. Die Stapelspitze wird gelöscht und der Stapelzeiger erhöht. Das Ergebnis ersetzt dann y und bildet die neue Stapelspitze.
Durch die Angabe des Verhältnisses y/x in zwei Registern ist es leicht möglich, mit den Befehlen FPTAN und FPATAN die anderen Winkelfunktionen sin, cos, arcsin und arccos mit Hilfe einfacher trigonometrischer Beziehungen zu berechnen.

F2XM1
Dieser Befehl berechnet die Funktion

$$y = 2^x - 1$$

Der Wert x wird aus der Stapelspitze genommen und muß im Bereich $0 \leq x < 0{,}5$ liegen. Das Ergebnis wird wieder in der Stapelspitze abgelegt.
Mit Hilfe der Befehle FLDL2T, FLDL2E und FYL2X können hieraus auch die Funktionen 10^x, e^x und y^x berechnet werden.

FYL2X
Dieser Befehl berechnet die Funktion

$$z = y \log_2 x$$

Der Wert x wird aus der Stapelspitze genommen und muß im Bereich $0 < x < \infty$ liegen. Der Wert y wird aus ST(1) genommen und muß im Bereich $-\infty < y < +\infty$ liegen. Die Stapelspitze wird gelöscht, der Stapelzeiger erhöht und das Ergebnis in der neuen Stapelspitze abgelegt.

FYL2XP1
Dieser Befehl berechnet die Funktion:

$$z = y \log_2 (x + 1)$$

Der Wert x wird aus der Stapelspitze genommen und muß im Bereich $0 < |x| < 1 - \sqrt{2}/2$ liegen. Der Wert y wird aus ST(1) genommen und muß im Bereich $-\infty < y < +\infty$ liegen. Die Stapelspitze wird gelöscht, der Stapelzeiger erhöht und das Ergebnis in der neuen Stapelspitze abgelegt.
Diese Funktion ermöglicht sehr genaue Berechnungen der Logarithmen von Zahlen, die sehr nahe bei 1 liegen, da der Operand x genauer dargestellt werden kann als 1+x.

7.3.5. Befehle zum Laden von Konstanten

Diese Befehle laden die Stapelspitze mit sehr genauen Werten üblicher Konstanten, die im K1810WM87 implementiert sind. Bei jedem Laden wird der Stapelzeiger um 1 erniedrigt, so daß die Konstanten in das erste freie Register geladen werden, welches dann die neue Stapelspitze bildet. Die Werte haben eine Genauigkeit von etwa 19 Dezimalstellen.
Es werden keine Operanden angegeben.

FLDZ lädt den Wert +0

FLD1 lädt den Wert +1

FLDPI lädt den Wert π

FLDL2T lädt den Wert $\log_2 10$

FLDL2E lädt den Wert $\log_2 e$

FLDLG2 lädt den Wert lg 2

FLDLN2 lädt den Wert ln 2

7.3.6. Befehle zur Prozessorsteuerung

Diese Befehle werden nicht für arithmetische Berechnungen benötigt. Sie dienen der Steuerung des K1810WM87 auf der Systemebene und erlauben die Initialisierung, Taskumschaltung, Fehlerbehandlung u.a.
Für die meisten dieser Befehle stehen zwei Mnemoniks zur Auswahl, eine normale und eine "NO-WAIT"-Form. Diese zweite Form veranlaßt den Assembler, keinen WAIT-Befehl vor dem K1810WM87-Befehl zu generieren (siehe auch Abschn. 7.2.3). Diese Form sollte in kritischen Programmzweigen angewendet werden, in denen Interrupts verboten sind, der Coprozessor aber einen Interrupt senden könnte (vgl. Abschn. 7.2.5). In normalen Anwenderprogrammen, in denen

Interrupts erlaubt sind, sollte die WAIT-Form angewendet werden. Das Problem tritt nicht auf, wenn alle Exceptions maskiert sind. Mit Ausnahme der Befehle FNSTENV und FNSAVE führt der K1810WM87 die "NO-WAIT"-Befehle parallel zu einer laufenden Operation aus, das heißt, sie werden auch bei aktiver BUSY-Leitung ausgeführt.

FINIT / FNINIT (Initialize)
Dieser Befehl initialisiert den K1810WM87. Dabei nimmt er den gleichen Zustand ein wie nach einem RESET-Signal (siehe Abschn. 7.2.6).

FDISI / FNDISI (Disable Interrupts)
Dieser Befehl setzt die Interruptmaske im Steuerwort auf 1.

FENI / FNENI (Enable Interrupts)

Dieser Befehl setzt die Interruptmaske im Steuerwort auf 0. Dadurch werden Interrupts des K1810WM87 erlaubt.

FLDCW src (Load Control Word)
Der Befehl FLDCW lädt das Steuerwort des K1810WM87 mit einem Datenwort aus dem Speicher. Dadurch kann ein gewünschter Modus eingestellt oder geändert werden. Dieser Befehl kann erst nach Beendigung der laufenden Operation ausgeführt werden.
Falls ein Exception-Flag gesetzt ist und mit FLDCW die Maske dafür zurückgesetzt wird und Interrupts freigegeben werden, so wird unmittelbar ein Interrupt generiert. Deshalb sollten Exceptions zurückgesetzt sein, bevor das Steuerwort geändert wird.

FSTCW / FNSTCW dest (Store Control Word)
Dieser Befehl überträgt das aktuelle Steuerwort in den Speicher.

FSTSW / FNSTSW dest (Store Status Word)
Dieser Befehl überträgt das aktuelle Statuswort in den Speicher.

FCLEX / FNCLEX (Clear Exceptions)
Der Befehl FCLEX setzt alle Exception-Flags, das Interruptanforderungsbit (IR) und das Busy-Bit (B) im Statuswort auf 0. Dadurch gehen auch die Signale INT und BUSY in den inaktiven Zustand.
Dieser Befehl muß in Interruptprozeduren zur Exceptionbehandlung ausgeführt werden, bevor in die K1810WM87-Prozedur zurückgekehrt wird. Andernfalls wird sofort ein neuer Interrupt generiert.

FSAVE / FNSAVE dest (Save State)
Der FSAVE-Befehl überträgt den gesamten Zustand des K1810WM87 in den Speicher. Die Inhalte der 8 Stapelregister, die Exception-Pointer, das TAG-, Status- und Steuerwort werden in ein 94 Byte großes Feld abgebildet (Bild 7.6). Anschließend wird der K1810WM87 initialisiert; er nimmt den gleichen Zustand wie nach RESET ein (s. Abschn. 7.2.6). Falls der FNSAVE-Befehl dekodiert wird, während eine Operation noch nicht beendet ist, wird die Ausführung zurückgestellt, bis der laufende Befehl vollständig ausgeführt wurde.

Normalerweise wird dieser Befehl benutzt, um den Zustand des K1810WM87 im Stapel des K1810WM86 zu retten, wenn eine höherpriorisierte Task die laufende Berechnung unterbrechen möchte. Anschließend kann problemlos wieder in das alte Programm zurückgekehrt werden.

FRSTOR src (Restore State)
Dieser Befehl lädt den K1810WM87 mit dem kompletten 94-Byte-Speicherabbild, das vorher mit dem FSAVE-Befehl angelegt wurde. Damit wird die Rückkehr in den Zustand ermöglicht, der vor einer Neuinitialisierung bestanden hat.

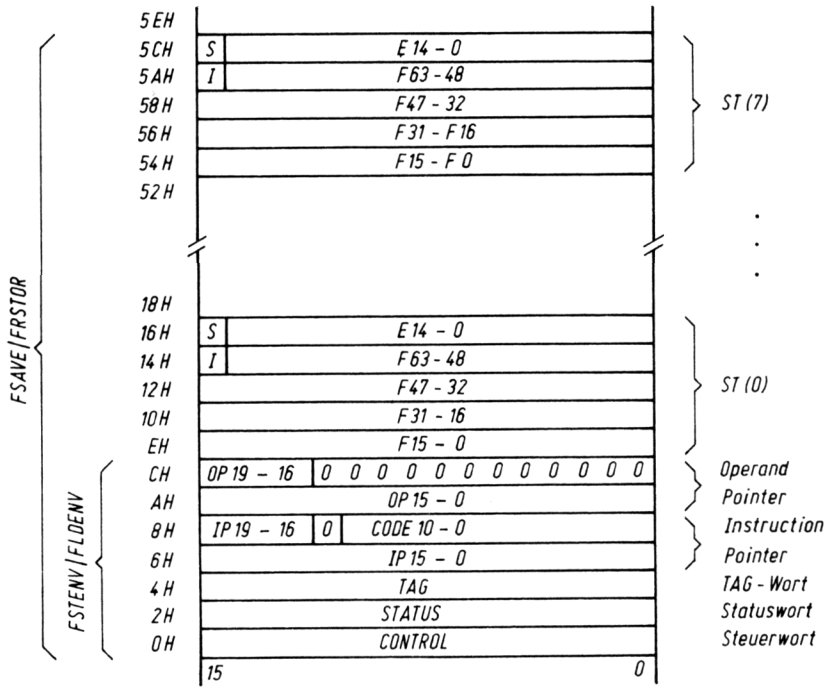


Bild 7.6. Speicherabbild des K1810WM87-Zustands

FSTENV / FNSTENV dest (Store Environment)
Der FSTENV-Befehl überträgt den Grundzustand des K1810WM87 in den Speicher. Dazu werden die Exception-Pointer, das TAG-, Status- und Steuerwort in ein 14-Byte-Feld übertragen (vgl. Bild 7.6). Anschließend werden alle Exceptions maskiert. (Die Interruptmaske bleibt unbeeinflußt.) Falls ein FNSTENV-Befehl dekodiert wird, wenn der K1810WM87 eine Operation noch nicht beendet hat, wird dieser Befehl solange zurückgestellt.
Dieser Befehl kann bei der Exception-Behandlung angewendet werden, da mit ihm außer auf den Status auch auf die Exception-Pointer zugegriffen werden kann.

FLDENV src (Load Environment)
Dieser Befehl lädt den K1810WM87 wieder mit dem Grundzustand, der durch eine vorangegangene FSTENV-Operation im Speicher abgebildet wurde.

FINCSTP (Increment Stack Pointer)
Dieser Befehl erhöht den Wert des Stapelzeigers um 1, ohne den Inhalt der Stapelregister oder des TAG-Wortes zu beeinflussen. (TOS = 7 wird zu TOS = 0 inkrementiert.)

FDECSTP (Decrement Stack Pointer)
Dieser Befehl erniedrigt den Wert des Stapelzeigers um 1, ohne den Inhalt der Stapelregister oder des TAG-Wortes zu beeinflussen. (TOS = 0 wird zu TOS = 7 dekrementiert.)

FFREE dest (Free Register)
Der TAG-Wert des spezifizierten Registers wird auf 11 gesetzt. Dadurch wird dieses Register als frei gekennzeichnet, ohne daß der Registerinhalt selbst beeinflußt wird.

FNOP (No Operation)
Der Befehl FNOP bewirkt eine Leeroperation; das heißt, der K1810WM87 tut nichts. (Intern wird die Operation FST ST,ST(0) ausgeführt.)

FWAIT (Wait)
Der FWAIT-Befehl wird nicht vom K1810WM87 ausgeführt. (Es ist kein ESC-Kode.) Der Befehl FWAIT bewirkt die Generierung eines WAIT-Befehls für den K1810WM86.
Diese alternative Mnemonik für WAIT sollte benutzt werden, wenn mit einem K1810WM87-Emulator gearbeitet werden soll. Dieser eliminiert nämlich diese Befehle im Gegensatz zur WAIT-Mnemonik (siehe auch Abschn. 7.2.3).

Anhang A: Befehle des K1810WM86

Die folgende Liste enthält eine Übersicht über alle Befehle des K1810WM86. Es werden Mnemoniks, anwendbare Operanden, Anzahl der Kodebytes, Ausführungszeiten (Anzahl der CPU-Takte) und Flagbeeinflussungen angegeben.

Anmerkungen:

1. Die Zahl der Kodebytes ist vom Adreßmodus abhängig.
2. Die Zahl der Buszyklen ist vom Operandentyp und der Operandenadresse abhängig.
3. EA ist die Zeit zur Berechnung der effektiven Adresse. Diese kann je nach Adreßmodus 5 bis 12 Takte betragen (siehe auch Tafel 5.3).
4. rp ist die Zahl der Wiederholungen bei Befehlen mit REP-Präfix;
   bt ist die Zahl der Verschiebungen bei Verschiebe- bzw. Rotationsbefehlen.
5. Die mit * gekennzeichneten Mnemoniks sind zur Information angegeben.
6. Flagbeeinflussung:
   - nicht beeinflußt
   - x je nach Ergebnis beeinflußt
   - u undefiniert
   - 1 gesetzt
   - 0 zurückgesetzt
   - r aus dem Stapel rekonstruiert

Mnemonik	Operanden	Bytes	Takte	Zyklen	Flags ODITSZAPC
AAA		1	4	-	u---uuxux
AAD		2	60	-	u---xxuxu
AAM		2	83	-	u---xxuxu
AAS		1	4	-	u---uuxux
ADC	reg,reg	2	3	-	x---xxxxx
	reg,mem	2-4	9+EA	1 (2)	x---xxxxx
	mem,reg	2-4	16+EA	2 (4)	x---xxxxx
	reg,data	3-4	4	-	x---xxxxx
	mem,data	3-6	17+EA	2 (4)	x---xxxxx
	ac,data	2-3	4	-	x---xxxxx
ADD	reg,reg	2	3	-	x---xxxxx
	reg,mem	2-4	9+EA	1 (2)	x---xxxxx
	mem,reg	2-4	16+EA	2 (4)	x---xxxxx
	reg,data	3-4	4	-	x---xxxxx
	mem,data	3-6	17+EA	2 (4)	x---xxxxx
	ac,data	2-3	4	-	x---xxxxx
AND	reg,reg	2	3	-	0---xxux0
	reg,mem	2-4	9+EA	1 (2)	0---xxux0
	mem,reg	2-4	16+EA	2 (4)	0---xxux0
	reg,data	3-4	4	-	0---xxux0
	mem,data	3-6	17+EA	2 (4)	0---xxux0
	ac,data	2-3	4	-	0---xxux0

Mnemonik	Operanden	Bytes	Takte	Zyklen	Flags ODITSZAPC
CALL	nearproc	3	19	1	---------
	farproc	5	28	2	---------
	mptr16	2-4	21+EA	2 (3)	---------
	mptr32	2-4	37+EA	4 (6)	---------
	regptr	2	16	1	---------
CBW		1	2	-	---------
CLC		1	2	-	--------0
CLD		1	2	-	-0-------
CLI		1	2	-	--0------
CMC		1	2	-	--------x
CMP	reg,reg	2	3	-	x---xxxxx
	reg,mem	2-4	9+EA	1 (2)	x---xxxxx
	mem,reg	2-4	9+EA	1 (2)	x---xxxxx
	reg,data	3-4	4	-	x---xxxxx
	mem,data	3-6	10+EA	1 (2)	x---xxxxx
	ac,data	2-3	4	-	x---xxxxx
CMPS	deststr,srcstr	1	22	2 (4)	x---xxxxx
CMPSB		1	22	2	x---xxxxx
CMPSW		1	22	2 (4)	x---xxxxx
*REPE CMPS		2	9+22/rp		x---xxxxx
*REPNE CMPS		2	9+22/rp		x---xxxxx
*REPNZ CMPS		2	9+22/rp		x---xxxxx
*REPZ CMPS		2	9+22/rp		x---xxxxx
CWD		1	5	-	---------
DAA		1	4	-	x---xxxxx
DAS		1	4	-	u---xxxxx
DEC	reg16	1	2	-	x---xxxx-
	reg8	2	3	-	x---xxxx-
	mem	2	15+EA	2 (4)	x---xxxx-
DIV	reg8	2	80- 90	-	u---uuuuu
	reg16	2	144-162	-	u---uuuuu
	mem8	2-4	86- 96+EA	1	u---uuuuu
	mem16	2-4	150-168+EA	1 (2)	u---uuuuu
ESC	code,reg	2	2	-	---------
	code,mem	2-4	8+EA	1 (2)	---------
HLT		1	2	-	---------
IDIV	reg8	2	101-112	-	u---uuuuu
	reg16	2	165-184	-	u---uuuuu
	mem8	2-4	107-118+EA	1	u---uuuuu
	mem16	2-4	171-190+EA	1 (2)	u---uuuuu
IMUL	reg8	2	80- 98	-	x---uuuux
	reg16	2	128-154	-	x---uuuux
	mem8	2-4	86-104+EA	1	x---uuuux
	mem16	2-4	134-160+EA	1 (2)	x---uuuux
IN	ac,port8	2	10	1 (2)	---------
	ac,DX	1	8	1 (2)	---------
INC	reg16	1	2	-	x---xxxx-
	reg8	2	3	-	x---xxxx-
	mem	2-4	15+EA	2 (4)	x---xxxx-

Mnemonik	Operanden	Bytes	Takte	Zyklen	Flags ODITSZAPC
INT	typ	2	51	5	--00-----
	3	1	52	5	--00-----
INTO		1	53/4	5	--00-----
IRET		1	24	3	rrrrrrrrr
JA	shortlab	2	16/4	-	---------
JAE	shortlab	2	16/4	-	---------
JB	shortlab	2	16/4	-	---------
JBE	shortlab	2	16/4	-	---------
JC	shortlab	2	16/4	-	---------
JCXZ	shortlab	2	18/6	-	---------
JE	shortlab	2	16/4	-	---------
JG	shortlab	2	16/4	-	---------
JGE	shortlab	2	16/4	-	---------
JL	shortlab	2	16/4	-	---------
JLE	shortlab	2	16/4	-	---------
JMP	shortlab	2	15	-	---------
	nearlab	3	15	-	---------
	farlab	5	15	-	---------
	mptr16	2-4	18+EA	1 (2)	---------
	mptr32	2-4	24+EA	2 (4)	---------
	regptr	2	11	-	---------
JNA	shortlab	2	16/4	-	---------
JNAE	shortlab	2	16/4	-	---------
JNB	shortlab	2	16/4	-	---------
JNBE	shortlab	2	16/4	-	---------
JNC	shortlab	2	16/4	-	---------
JNE	shortlab	2	16/4	-	---------
JNG	shortlab	2	16/4	-	---------
JNGE	shortlab	2	16/4	-	---------
JNL	shortlab	2	16/4	-	---------
JNLE	shortlab	2	16/4	-	---------
JNO	shortlab	2	16/4	-	---------
JNP	shortlab	2	16/4	-	---------
JNS	shortlab	2	16/4	-	---------
JNZ	shortlab	2	16/4	-	---------
JO	shortlab	2	16/4	-	---------
JP	shortlab	2	16/4	-	---------
JPE	shortlab	2	16/4	-	---------
JPO	shortlab	2	16/4	-	---------
JS	shortlab	2	16/4	-	---------
JZ	shortlab	2	16/4	-	---------
LAHF		1	4	-	---------
LDS	reg16,mptr32	2-4	16+EA	2 (4)	---------
LEA	reg16,mem	2-4	2+EA	-	---------
LES	reg16,mptr32	2-4	16+EA	2 (4)	---------
LOCK	(PREFIX)	1	2	-	---------
LODS	srcstr	1	12	1 (2)	---------
LODSB		1	12	1	---------
LODSW		1	12	1 (2)	---------

Mnemonik	Operanden	Bytes	Takte	Zyklen	Flags ODITSZAPC
LOOP	shortlab	2	17/5	-	---------
LOOPE	shortlab	2	18/6	-	---------
LOOPNE	shortlab	2	19/5	-	---------
LOOPNZ	shortlab	2	19/5	-	---------
LOOPZ	shortlab	2	18/6	-	---------
MOV	reg,reg	2	2	-	---------
	reg,mem	2-4	8+EA	1 (2)	---------
	mem,reg	2-4	9+EA	1 (2)	---------
	ac,mem	3	10	1 (2)	---------
	mem,ac	3	10	1 (2)	---------
	reg,data	2-3	4	-	---------
	mem,data	3-6	10+EA	1 (2)	---------
	segreg,reg16	2	2	-	---------
	reg16,segreg	2	2	-	---------
	segreg,mem16	2-4	8+EA	1 (2)	---------
	mem16,segreg	2-4	9+EA	1 (2)	---------
MOVS	deststr,srcstr	1	18	2 (4)	---------
MOVSB		1	18	2	---------
MOVSW		1	18	2 (4)	---------
*REP MOVS		2	9+17/rp		---------
MUL	reg8	2	70- 77	-	x---uuuux
	reg16	2	118-133	-	x---uuuux
	mem8	2-4	76- 83+EA	1	x---uuuux
	mem16	2-4	124-139+EA	1 (2)	x---uuuux
NEG	reg	2	3	-	x---xxxxx
	mem	2-4	16+EA	2 (4)	x---xxxxx
NOP		1	3	-	---------
NOT	reg	2	3	-	---------
	mem	2-4	16+EA	2 (4)	---------
OR	reg,reg	2	3	-	0---xxux0
	reg,mem	2-4	9+EA	1 (2)	0---xxux0
	mem,reg	2-4	16+EA	2 (4)	0---xxux0
	reg,data	3-4	4	-	0---xxux0
	mem,data	3-6	17+EA	2 (4)	0---xxux0
	ac,data	2-3	4	-	0---xxux0
OUT	port8,ac	2	10	1 (2)	---------
	DX,ac	1	8	1 (2)	---------
POP	reg16	1	8	1	---------
	mem16	2-4	17+EA	2 (3)	---------
	segreg	1	8	1	---------
POPF		1	8	1	rrrrrrrrr
PUSH	reg16	1	11	1	---------
	mem16	2-4	16+EA	2 (3)	---------
	segreg	1	10	1	---------
PUSHF		1	10	1	---------
RCL	reg,1	2	2	-	x-------x
	reg,CL	2	8+4/bt	-	x-------x
	mem,1	2-4	15+EA	2 (4)	x-------x
	mem,CL	2-4	20+4/bt+EA	2 (4)	x-------x

Mnemonik	Operanden	Bytes	Takte	Zyklen	Flags ODITSZAPC
RCR	reg,1	2	2	-	x-------x
	reg,CL	2	8+4/bt	-	x-------x
	mem,1	2-4	15+EA	2 (4)	x-------x
	mem,CL	2-4	20+4/bt+EA	2 (4)	x-------x
REP	(PREFIX)	1	2	-	---------
REPE	(PREFIX)	1	2	-	---------
REPNE	(PREFIX)	1	2	-	---------
REPNZ	(PREFIX)	1	2	-	---------
REPZ	(PREFIX)	1	2	-	---------
RET (near)		1	8	1	---------
(near)	popvalue	3	12	1	---------
(far)		1	18	2	---------
(far)	popvalue	3	17	2	---------
ROL	reg,1	2	2	-	x-------x
	reg,CL	2	8+4/bt	-	x-------x
	mem,1	2-4	15+EA	2 (4)	x-------x
	mem,CL	2-4	20+4/bt+EA	2 (4)	x-------x
ROR	reg,1	2	2	-	x-------x
	reg,CL	2	8+4/bt	-	x-------x
	mem,1	2-4	15+EA	2 (4)	x-------x
	mem,CL	2-4	20+4/bt+EA	2 (4)	x-------x
SAHF		1	4	-	----rrrrr
SAL	reg,1	2	2	-	x---xxuxx
	reg,CL	2	8+4/bt	-	x---xxuxx
	mem,1	2-4	15+EA	2 (4)	x---xxuxx
	mem,CL	2-4	20+4/bt+EA	2 (4)	x---xxuxx
SAR	reg,1	2	2	-	x---xxuxx
	reg,CL	2	8+4/bt	-	x---xxuxx
	mem,1	2-4	15+EA	2 (4)	x---xxuxx
	mem,CL	2-4	20+4/bt+EA	2 (4)	x---xxuxx
SBB	reg,reg	2	3	-	x---xxxxx
	reg,mem	2-4	9+EA	1 (2)	x---xxxxx
	mem,reg	2-4	16+EA	2 (4)	x---xxxxx
	reg,data	3-4	4	-	x---xxxxx
	mem,data	3-6	17+EA	2 (4)	x---xxxxx
	ac,data	2-3	4	-	x---xxxxx
SCAS	deststr	1	15	1 (2)	x---xxxxx
SCASB		1	15	1	x---xxxxx
SCASW		1	15	1 (2)	x---xxxxx
*REPE SCAS		2	9+15/rp		x---xxxxx
*REPNE SCAS		2	9+15/rp		x---xxxxx
*REPNZ SCAS		2	9+15/rp		x---xxxxx
*REPZ SCAS		2	9+15/rp		x---xxxxx
(SEG OVERRIDE PREFIX)		1	2	-	---------
SHL	reg,1	2	2	-	x---xxuxx
	reg,CL	2	8+4/bt	-	x---xxuxx
	mem,1	2-4	15+EA	2 (4)	x---xxuxx
	mem,CL	2-4	20+4/bt+EA	2 (4)	x---xxuxx

Mnemonik	Operanden	Bytes	Takte	Zyklen	Flags ODITSZAPC
SHR	reg,1	2	2	-	x---xxuxx
	reg,CL	2	8+4/bt	-	x---xxuxx
	mem,1	2-4	15+EA	2 (4)	x---xxuxx
	mem,CL	2-4	20+4/bt+EA	2 (4)	x---xxuxx
STC		1	2	-	--------1
STD		1	2	-	-1-------
STI		1	2	-	--1------
STOS	deststr	1	11	1 (2)	---------
STOSB		1	11	1	---------
STOSW		1	11	1 (2)	---------
*REP STOS		2	9+10/rp	1/rp	---------
SUB	reg,reg	2	3	-	x---xxxxx
	reg,mem	2-4	9+EA	1 (2)	x---xxxxx
	mem,reg	2-4	16+EA	2 (4)	x---xxxxx
	reg,data	3-4	4	-	x---xxxxx
	mem,data	3-6	17+EA	2 (4)	x---xxxxx
	ac,data	2-3	4	-	x---xxxxx
TEST	reg,reg	2	3	-	0---xxux0
	reg,mem	2-4	9+EA	1 (2)	0---xxux0
	reg,data	3-4	5	-	0---xxux0
	mem,data	3-6	11+EA	1 (2)	0---xxux0
	ac,data	2-3	4	-	0---xxux0
WAIT		1	3+5n	-	---------
XCHG	reg,reg	2	4	-	---------
	mem,reg	2-4	17+EA	2 (4)	---------
	ac,reg16	1	3	-	---------
XLAT	(tab)	1	11	1	---------
XOR	reg,reg	2	3	-	0---xxux0
	reg,mem	2-4	9+EA	1 (2)	0---xxux0
	mem,reg	2-4	16+EA	2 (4)	0---xxux0
	reg,data	3-4	4	-	0---xxux0
	mem,data	3-6	17+EA	2 (4)	0---xxux0
	ac,data	2-3	4	-	0---xxux0

Anhang B: Befehle des K1810WM87

Die Liste enthält eine Übersicht über alle Befehle des Numerikcoprozessors K1810WM87. Sie enthält folgende Informationen:

Spalte 1: Befehlsmnemonik (Mnem.);
Spalte 2: anwendbare Operanden;
Spalte 3: Anzahl der Kodebytes (B.);
Spalte 4: Ausführungszeit als Anzahl der CPU-Takte;
Spalte 5: Anzahl der Buszyklen (Zyk.)
Spalte 6: Exceptions, die geprüft werden und auftreten können;
Spalte 7: Maschinenkode (ESC-Kode).

Verwendete Operandensymbole:

Symbol	Bedeutung
ST	Stapelspitze
ST(i)	i-tes Register unter der Stapelspitze
short real	32-Bit-Speichervariable im Short-Real-Format
long real	64-Bit-Speichervariable im Long-Real-Format
temp real	80-Bit-Speichervariable im Temporary-Real-Format
word integ	16-Bit-Speichervariable im Word-Integer-Format
short integ	32-Bit-Speichervariable im Short-Integer-Format
long integ	64-Bit-Speichervariable im Long-Integer-Format
decimal	80-Bit-Speichervariable im BCD-Format
mem	Speichervariable (allgemein, Wort oder Feld)

Befehlsformat:

11011 xxx	MD yyy R/M	(disp-lo)	(disp-hi)

ESC CODE 10 - 0

MD = 11: R/M = STi
MD = 00: mem, ohne disp
MD = 01: mem, disp8
MD = 10: mem, disp16

STi	ST(i)
000	ST
001	ST(1)
010	ST(2)
011	ST(3)
100	ST(4)
101	ST(5)
110	ST(6)
111	ST(7)

Anmerkungen:

1. Die Anzahl der Kodebytes ist abhängig vom Adreßmodus.
2. Die Ausführungszeit ist als typischer Wert angegeben. Abweichungen (im Mittel ±5 Takte, in Extremfällen mehr) können in Abhängigkeit vom Operandenwert, Exceptions, dem Busaustausch mit der Host-CPU und WAIT-Zuständen auftreten.
3. Die Anzahl der Buszyklen ist vom Operandentyp abhängig. Die angegebenen Werte schließen den CPU-Zyklus (Dummy Read) mit ein.
4. Der Maschinenkode ist vom Adreßmodus abhängig. Angegeben ist nur CODE 10-0. Für Speicheroperanden können alle Adressierungsarten des K1810WM86 angewendet werden (EA).
   Der mit * gekennzeichnete Kode für FWAIT ist kein ESC-Kode.

Mnem.	Operanden	B.	Takte (typ.)	Zyk.	Excpt. PUOZDI	Kode xxx MDyyyR/M
FABS		2	14	-	-----x	001 11100001
FADD	ST,ST(i)	2	85	-	xxx-xx	000 11000STi
	ST(i),ST	2	85	-	xxx-xx	100 11000STi
	short real	2-4	105+EA	2 (4)	xxx-xx	000 MD000R/M
	long real	2-4	110+EA	4 (6)	xxx-xx	100 MD000R/M
FADDP	ST(i),ST	2	90	-	xxx-xx	110 11000STi
FBLD	decimal	2-4	300+EA	5 (7)	-----x	111 MD100R/M
FBSTP	decimal	2-4	530+EA	6 (8)	-----x	111 MD110R/M
FCHS		2	15	-	-----x	001 11100000
FCLEX		2	5	-	------	011 11100010
FCOM	ST(i)	2	45	-	----xx	000 11010STi
	short real	2-4	65+EA	2 (4)	----xx	000 MD010R/M
	long real	2-4	70+EA	4 (6)	----xx	100 MD010R/M
FCOMP	ST(i)	2	47	-	----xx	000 11011STi
	short real	2-4	68+EA	2 (4)	----xx	000 MD011R/M
	long real	2-4	72+EA	4 (6)	----xx	100 MD011R/M
FCOMPP		2	50	-	----xx	110 11011001
FDECST		2	9	-	------	001 11110110
FDISI		2	5	-	------	011 11100001
FDIV	ST,ST(i)	2	198	-	xxxxxx	000 11110STi
	ST(i),ST	2	198	-	xxxxxx	100 11111STi
	short real	2-4	220+EA	2 (4)	xxxxxx	000 MD110R/M
	long real	2-4	225+EA	4 (6)	xxxxxx	100 MD110R/M
FDIVP	ST(i),ST	2	202	-	xxxxxx	110 11111STi
FDIVR	ST,ST(i)	2	199	-	xxxxxx	000 11111STi
	ST(i),ST	2	199	-	xxxxxx	100 11110STi
	short real	2-4	221+EA	2 (4)	xxxxxx	000 MD111R/M
	long real	2-4	226+EA	4 (6)	xxxxxx	100 MD111R/M
FDIVRP	ST(i),ST	2	203	-	xxxxxx	110 11110STi
FENI		2	5	-	------	011 11100000
FFREE	ST(i)	2	11	-	------	101 11000STi
FIADD	word integ	2-4	120+EA	1 (2)	x-x-xx	110 MD000R/M
	short integ	2-4	125+EA	2 (4)	x-x-xx	010 MD000R/M
FICOM	word integ	2-4	80+EA	1 (2)	----xx	110 MD010R/M
	short integ	2-4	85+EA	2 (4)	----xx	010 MD010R/M
FICOMP	word integ	2-4	82+EA	1 (2)	----xx	110 MD011R/M
	short integ	2-4	87+EA	2 (4)	----xx	010 MD011R/M
FIDIV	word integ	2-4	230+EA	1 (2)	xxxxxx	110 MD110R/M
	short integ	2-4	236+EA	2 (4)	xxxxxx	010 MD110R/M
FIDIVR	word integ	2-4	230+EA	1 (2)	xxxxxx	110 MD111R/M
	short integ	2-4	237+EA	2 (4)	xxxxxx	010 MD111R/M
FILD	word integ	2-4	50+EA	1 (2)	-----x	111 MD000R/M
	short integ	2-4	56+EA	2 (4)	-----x	011 MD000R/M
	long integ	2-4	64+EA	4 (6)	-----x	111 MD101R/M
FIMUL	word integ	2-4	130+EA	1 (2)	x-x-xx	110 MD001R/M
	short integ	2-4	136+EA	2 (4)	x-x-xx	010 MD001R/M
FINCST		2	9	-	------	001 11110111
FINIT		2	5	-	------	011 11100011

Mnem.	Operanden	B.	Takte (typ.)	Zyk.	Excpt. PUOZDI	Kode xxx MDyyyR/M
FIST	word integ	2-4	86+EA	2 (4)	x----x	111 MD010R/M
	short integ	2-4	88+EA	3 (5)	x----x	011 MD010R/M
FISTP	word integ	2-4	88+EA	2 (4)	x----x	111 MD011R/M
	short integ	2-4	90+EA	3 (5)	x----x	011 MD011R/M
	long integ	2-4	100+EA	5 (7)	x----x	111 MD111R/M
FISUB	word integ	2-4	120+EA	1 (2)	x-x-xx	110 MD100R/M
	short integ	2-4	125+EA	2 (4)	x-x-xx	010 MD100R/M
FISUBR	word integ	2-4	120+EA	1 (2)	x-x-xx	110 MD101R/M
	short integ	2-4	125+EA	2 (4)	x-x-xx	010 MD101R/M
FLD	ST(i)	2	20	-	----xx	001 11000STi
	short real	2-4	43+EA	2 (4)	----xx	001 MD000R/M
	long real	2-4	46+EA	4 (6)	----xx	101 MD000R/M
	temp real	2-4	57+EA	5 (7)	----xx	011 MD101R/M
FLDCW	mem(2bytes)	2-4	10+EA	1 (2)	------	001 MD101R/M
FLDENV	mem(14bytes)	2-4	40+EA	7 (9)	------	001 MD100R/M
FLDLG2		2	21	-	-----x	001 11101100
FLDLN2		2	20	-	-----x	001 11101101
FLDL2E		2	18	-	-----x	001 11101010
FLDL2T		2	19	-	-----x	001 11101001
FLDPI		2	19	-	-----x	001 11101011
FLDZ		2	14	-	-----x	001 11101110
FLD1		2	18	-	-----x	001 11101000
FMUL	ST,ST(i)	2	138	-	xxx-xx	000 11001STi
	ST(i),ST	2	138	-	xxx-xx	100 11001STi
	short real	2-4	118+EA	2 (4)	xxx-xx	000 MD001R/M
	long real	2-4	161+EA	4 (6)	xxx-xx	100 MD001R/M
FMULP	ST(i),ST	2	142	-	xxx-xx	110 11001STi
FNCLEX		2	5	-	------	011 11100010
FNDISI		2	5	-	------	011 11100001
FNOP		2	13	-	------	001 11010000
FNSAVE	mem(94bytes)	2-4	210+EA	8 (50)	------	101 MD110R/M
FNSTCW	mem(2bytes)	2-4	15+EA	2 (4)	------	001 MD111R/M
FNSTENV	mem(14bytes)	2-4	45+EA	8 (10)	------	001 MD110R/M
FNSTSW	mem(2bytes)	2-4	15+EA	2 (4)	------	101 MD111R/M
FPATAN		2	650	-	xx----	001 11110011
FPREM		2	125	-	-x--xx	001 11111000
FPTAN		2	450	-	x----x	001 11110010
FRNDINT		2	45	-	x----x	001 11111100
FRSTOR	mem(94bytes)	2-4	210+EA	7 (49)	------	101 MD100R/M
FSAVE	mem(94bytes)	2-4	210+EA	8 (50)	------	101 MD110R/M
FSCALE		2	35	-	-xx--x	001 11111101
FSQRT		2	183	-	x---xx	001 11111010
FST	ST(i)	2	18	-	xxx--x	101 11010STi
	short real	2-4	87+EA	3 (5)	xxx--x	001 MD010R/M
	long real	2-4	100+EA	5 (7)	xxx--x	101 MD010R/M
FSTCW	mem(2bytes)	2-4	15+EA	2 (4)	------	001 MD111R/M
FSTENV	mem(14bytes)	2-4	45+EA	8 (10)	------	001 MD110R/M

Mnem.	Operanden	B.	Takte (typ.)	Zyk.	Excpt. PUOZDI	Kode xxx MDyyyR/M
FSTP	ST(i)	2	20	-	xxx--x	101 11011STi
	short real	2-4	89+EA	3 (5)	xxx--x	001 MD011R/M
	long real	2-4	102+EA	5 (7)	xxx--x	101 MD011R/M
	temp real	2-4	55+EA	6 (8)	xxx--x	011 MD111R/M
FSTSW	mem(2bytes)	2-4	15+EA	2 (4)	------	101 MD111R/M
FSUB	ST,ST(i)	2	85	-	xxx-xx	000 11100STi
	ST(i),ST	2	85	-	xxx-xx	100 11101STi
	short real	2-4	105+EA	2 (4)	xxx-xx	000 MD100R/M
	long real	2-4	110+EA	4 (6)	xxx-xx	100 MD100R/M
FSUBP	ST(i),ST	2	90	-	xxx-xx	110 11101STi
FSUBR	ST,ST(i)	2	87	-	xxx-xx	000 11101STi
	ST(i),ST	2	87	-	xxx-xx	100 11100STi
	short real	2-4	105+EA	2 (4)	xxx-xx	000 MD101R/M
	long real	2-4	110+EA	4 (6)	xxx-xx	100 MD101R/M
FSUBPR	ST(i),ST	2	90	-	xxx-xx	110 11100STi
FTST		2	42	-	----xx	001 11100100
FWAIT		1	3+5n	-	------	* 10011011 *
FXAM		2	17	-	------	001 11100101
FXCH	ST(i)	2	12	-	-----x	001 11001STi
FXTRACT		2	50	-	-----x	001 11110100
FYL2X		2	950	-	x-----	001 11110001
FYL2XP1		2	850	-	x-----	001 11111001
F2XM1		2	500	-	xx----	001 11110000

Anhang C: Maschinenbefehle in numerischer Reihenfolge

Kodebytes 1	2	3 bis 6	Befehl	
00	MD REG R/M	**	ADD	reg8/mem8,reg8
01	MD REG R/M	**	ADD	reg16/mem16,reg16
02	MD REG R/M	**	ADD	reg8,reg8/mem8
03	MD REG R/M	**	ADD	reg16,reg16/mem16
04	data		ADD	AL,data8
05	data-lo	data-hi	ADD	AX,data16
06			PUSH	ES
07			POP	ES
08	MD REG R/M	**	OR	reg8/mem8,reg8
09	MD REG R/M	**	OR	reg16/mem16,reg16
0A	MD REG R/M	**	OR	reg8,reg8/mem8
0B	MD REG R/M	**	OR	reg16,reg16/mem16
0C	data		OR	AL,data8
0D	data-lo	data-hi	OR	AX,data16
0E			PUSH	CS
0F			-	
10	MD REG R/M	**	ADC	reg8/mem8,reg8
11	MD REG R/M	**	ADC	reg16/mem16,reg16
12	MD REG R/M	**	ADC	reg8,reg8/mem8
13	MD REG R/M	**	ADC	reg16,reg16/mem16
14	data		ADC	AL,data8
15	data-lo	data-hi	ADC	AX,data16
16			PUSH	SS
17			POP	SS
18	MD REG R/M	**	SBB	reg8/mem8,reg8
19	MD REG R/M	**	SBB	reg16/mem16,reg16
1A	MD REG R/M	**	SBB	reg8,reg8/mem8
1B	MD REG R/M	**	SBB	reg16,reg16/mem16
1C	data		SBB	AL,data8
1D	data-lo	data-hi	SBB	AX,data16
1E			PUSH	DS
1F			POP	DS
20	MD REG R/M	**	AND	reg8/mem8,reg8
21	MD REG R/M	**	AND	reg16/mem16,reg16
22	MD REG R/M	**	AND	reg8,reg8/mem8
23	MD REG R/M	**	AND	reg16,reg16/mem16
24	data		AND	AL,data8
25	data-lo	data-hi	AND	AX,data16
26			ES:	
27			DAA	
28	MD REG R/M	**	SUB	reg8/mem8,reg8
29	MD REG R/M	**	SUB	reg16/mem16,reg16
2A	MD REG R/M	**	SUB	reg8,reg8/mem8
2B	MD REG R/M	**	SUB	reg16,reg16/mem16
2C	data		SUB	AL,data8
2D	data-lo	data-hi	SUB	AX,data16
2E			CS:	
2F			DAS	

Kodebytes 1	2	3 bis 6	Befehl	
30	MD REG R/M	**	XOR	reg8/mem8,reg8
31	MD REG R/M	**	XOR	reg16/mem16,reg16
32	MD REG R/M	**	XOR	reg8,reg8/mem8
33	MD REG R/M	**	XOR	reg16,reg16/mem16
34	data		XOR	AL,data8
35	data-lo	data-hi	XOR	AX,data16
36			SS:	
37			AAA	
38	MD REG R/M	**	CMP	reg8/mem8,reg8
39	MD REG R/M	**	CMP	reg16/mem16,reg16
3A	MD REG R/M	**	CMP	reg8,reg8/mem8
3B	MD REG R/M	**	CMP	reg16,reg16/mem16
3C	data		CMP	AL,data8
3D	data-lo	data-hi	CMP	AX,data16
3E			DS:	
3F			AAS	
40			INC	AX
41			INC	CX
42			INC	DX
43			INC	BX
44			INC	SP
45			INC	BP
46			INC	SI
47			INC	DI
48			DEC	AX
49			DEC	CX
4A			DEC	DX
4B			DEC	BX
4C			DEC	SP
4D			DEC	BP
4E			DEC	SI
4F			DEC	DI
50			PUSH	AX
51			PUSH	CX
52			PUSH	DX
53			PUSH	BX
54			PUSH	SP
55			PUSH	BP
56			PUSH	SI
57			PUSH	DI
58			POP	AX
59			POP	CX
5A			POP	DX
5B			POP	BX
5C			POP	SP
5D			POP	BP
5E			POP	SI
5F			POP	DI
60 bis 6F			-	

Kodebytes 1	2	3 bis 6	Befehl	
70	IP-disp		JO	shortlab
71	IP-disp		JNO	shortlab
72	IP-disp		JB/JNAE/JC	shortlab
73	IP-disp		JNB/JAE/JNC	shortlab
74	IP-disp		JE/JZ	shortlab
75	IP-disp		JNE/JNZ	shortlab
76	IP-disp		JBE/JNA	shortlab
77	IP-disp		JNBE/JA	shortlab
78	IP-disp		JS	shortlab
79	IP-disp		JNS	shortlab
7A	IP-disp		JP/JPE	shortlab
7B	IP-disp		JNP/JPO	shortlab
7C	IP-disp		JL/JNGE	shortlab
7D	IP-disp		JNL/JGE	shortlab
7E	IP-disp		JLE/JNG	shortlab
7F	IP-disp		JNLE/JG	shortlab
80	MD 000 R/M	** , data	ADD	reg8/mem8,data8
80	MD 001 R/M	** , data	OR	reg8/mem8,data8
80	MD 010 R/M	** , data	ADC	reg8/mem8,data8
80	MD 011 R/M	** , data	SBB	reg8/mem8,data8
80	MD 100 R/M	** , data	AND	reg8/mem8,data8
80	MD 101 R/M	** , data	SUB	reg8/mem8,data8
80	MD 110 R/M	** , data	XOR	reg8/mem8,data8
80	MD 111 R/M	** , data	CMP	reg8/mem8,data8
81	MD 000 R/M	** , data-lo,data-hi	ADD	reg16/mem16,data16
81	MD 001 R/M	** , data-lo,data-hi	OR	reg16/mem16,data16
81	MD 010 R/M	** , data-lo,data-hi	ADC	reg16/mem16,data16
81	MD 011 R/M	** , data-lo,data-hi	SBB	reg16/mem16,data16
81	MD 100 R/M	** , data-lo,data-hi	AND	reg16/mem16,data16
81	MD 101 R/M	** , data-lo,data-hi	SUB	reg16/mem16,data16
81	MD 110 R/M	** , data-lo,data-hi	XOR	reg16/mem16,data16
81	MD 111 R/M	** , data-lo,data-hi	CMP	reg16/mem16,data16
82	MD 000 R/M	** , data	ADD	reg8/mem8,data8
82	MD 010 R/M	** , data	ADC	reg8/mem8,data8
82	MD 011 R/M	** , data	SBB	reg8/mem8,data8
82	MD 101 R/M	** , data	SUB	reg8/mem8,data8
82	MD 111 R/M	** , data	CMP	reg8/mem8,data8
83	MD 000 R/M	** , data	ADD	reg16/mem16,data8
83	MD 010 R/M	** , data	ADC	reg16/mem16,data8
83	MD 011 R/M	** , data	SBB	reg16/mem16,data8
83	MD 101 R/M	** , data	SUB	reg16/mem16,data8
83	MD 111 R/M	** , data	CMP	reg16/mem16,data8
84	MD REG R/M	**	TEST	reg8/mem8,reg8
85	MD REG R/M	**	TEST	reg16/mem16,reg16
86	MD REG R/M	**	XCHG	reg8/mem8,reg8
87	MD REG R/M	**	XCHG	reg16/mem16,reg16
88	MD REG R/M	**	MOV	reg8/mem8,reg8
89	MD REG R/M	**	MOV	reg16/mem16,reg16
8A	MD REG R/M	**	MOV	reg8,reg8/mem8
8B	MD REG R/M	**	MOV	reg16,reg16/mem16
8C	MD OSR R/M	**	MOV	reg16/mem16,segreg

Kodebytes 1	2	3 bis 6	Befehl	
8D	MD REG R/M	**	LEA	reg16,mem16
8E	MD OSR R/M	**	MOV	segreg,reg16/mem16
8F	MD 000 R/M	**	POP	mem16
90			NOP	
91			XCHG	AX,CX
92			XCHG	AX,DX
93			XCHG	AX,BX
94			XCHG	AX,SP
95			XCHG	AX,BP
96			XCHG	AX,SI
97			XCHG	AX,DI
98			CBW	
99			CWD	
9A	IP-lo	IP-hi,CS-lo,CS-hi	CALL	farproc
9B			WAIT	
9C			PUSHF	
9D			POPF	
9E			SAHF	
9F			LAHF	
A0	adr-lo	adr-hi	MOV	AL,mem8
A1	adr-lo	adr-hi	MOV	AX,mem16
A2	adr-lo	adr-hi	MOV	mem8,AL
A3	adr-lo	adr-hi	MOV	mem16,AX
A4			MOVS	deststr8,srcstr8
A5			MOVS	deststr16,srcstr16
A6			CMPS	deststr8,srcstr8
A7			CMPS	deststr16,srcstr16
A8	data		TEST	AL,data8
A9	data-lo	data-hi	TEST	AX,data16
AA			STOS	deststr8
AB			STOS	deststr16
AC			LODS	srcstr8
AD			LODS	srcstr16
AE			SCAS	deststr8
AF			SCAS	deststr16
B0	data		MOV	AL,data8
B1	data		MOV	CL,data8
B2	data		MOV	DL,data8
B3	data		MOV	BL,data8
B4	data		MOV	AH,data8
B5	data		MOV	CH,data8
B6	data		MOV	DH,data8
B7	data		MOV	BH,data8
B8	data-lo	data-hi	MOV	AX,data16
B9	data-lo	data-hi	MOV	CX,data16
BA	data-lo	data-hi	MOV	DX,data16
BB	data-lo	data-hi	MOV	BX,data16
BC	data-lo	data-hi	MOV	SP,data16
BD	data-lo	data-hi	MOV	BP,data16
BE	data-lo	data-hi	MOV	SI,data16
BF	data-lo	data-hi	MOV	DI,data16

Kodebytes 1	2	3 bis 6	Befehl
C0			-
C1			-
C2	data-lo	data-hi	RET popvalue (near)
C3			RET (near)
C4	MD REG R/M	**	LES reg16,mem16
C5	MD REG R/M	**	LDS reg16,mem16
C6	MD 000 R/M	** , data	MOV mem8,data8
C7	MD 000 R/M	** , data-lo,data-hi	MOV mem16,data16
C8			-
C9			-
CA	data-lo	data-hi	RET popvalue (far)
CB			RET (far)
CC			INT 3
CD	data		INT typ
CE			INTO
CF			IRET
D0	MD 000 R/M	**	ROL reg8/mem8,1
D0	MD 001 R/M	**	ROR reg8/mem8,1
D0	MD 010 R/M	**	RCL reg8/mem8,1
D0	MD 011 R/M	**	RCR reg8/mem8,1
D0	MD 100 R/M	**	SAL/SHL reg8/mem8,1
D0	MD 101 R/M	**	SHR reg8/mem8,1
D0	MD 111 R/M	**	SAR reg8/mem8,1
D1	MD 000 R/M	**	ROL reg16/mem16,1
D1	MD 001 R/M	**	ROR reg16/mem16,1
D1	MD 010 R/M	**	RCL reg16/mem16,1
D1	MD 011 R/M	**	RCR reg16/mem16,1
D1	MD 100 R/M	**	SAL/SHL reg16/mem16,1
D1	MD 101 R/M	**	SHR reg16/mem16,1
D1	MD 111 R/M	**	SAR reg16/mem16,1
D2	MD 000 R/M	**	ROL reg8/mem8,CL
D2	MD 001 R/M	**	ROR reg8/mem8,CL
D2	MD 010 R/M	**	RCL reg8/mem8,CL
D2	MD 011 R/M	**	RCR reg8/mem8,CL
D2	MD 100 R/M	**	SAL/SHL reg8/mem8,CL
D2	MD 101 R/M	**	SHR reg8/mem8,CL
D2	MD 111 R/M	**	SAR reg8/mem8,CL
D3	MD 000 R/M	**	ROL reg16/mem16,CL
D3	MD 001 R/M	**	ROR reg16/mem16,CL
D3	MD 010 R/M	**	RCL reg16/mem16,CL
D3	MD 011 R/M	**	RCR reg16/mem16,CL
D3	MD 100 R/M	**	SAL/SHL reg16/mem16,CL
D3	MD 101 R/M	**	SHR reg16/mem16,CL
D3	MD 111 R/M	**	SAR reg16/mem16,CL
D4	0A		AAM
D5	0A		AAD
D6			-
D7			XLAT

Kodebytes 1	2	3 bis 6	Befehl	
D8	MD 000 R/M	**	FADD	short real
D8	MD 001 R/M	**	FMUL	short real
D8	MD 010 R/M	**	FCOM	short real
D8	MD 011 R/M	**	FCOMP	short real
D8	MD 100 R/M	**	FSUB	short real
D8	MD 101 R/M	**	FSUBR	short real
D8	MD 110 R/M	**	FDIV	short real
D8	MD 111 R/M	**	FDIVR	short real
D8	11 000 R/M		FADD	ST,ST(i)
D8	11 001 R/M		FMUL	ST,ST(i)
D8	11 010 R/M		FCOM	ST(i)
D8	11 011 R/M		FCOMP	ST(i)
D8	11 100 R/M		FSUB	ST,ST(i)
D8	11 101 R/M		FSUBR	ST,ST(i)
D8	11 110 R/M		FDIV	ST,ST(i)
D8	11 111 R/M		FDIVR	ST,ST(i)
D9	MD 000 R/M	**	FLD	short real
D9	MD 010 R/M	**	FST	short real
D9	MD 011 R/M	**	FSTP	short real
D9	MD 100 R/M	**	FLDENV	mem
D9	MD 101 R/M	**	FLDCW	mem
D9	MD 110 R/M	**	FSTENV	mem
D9	MD 111 R/M	**	FSTCW	mem
D9	11 000 R/M		FLD	ST(i)
D9	11 001 R/M		FXCH	ST(i)
D9	11 010 000		FNOP	
D9	11 011 R/M		FSTP	ST(i) +
D9	11 100 000		FCHS	
D9	11 100 001		FABS	
D9	11 100 100		FTST	
D9	11 100 101		FXAM	
D9	11 101 000		FLD1	
D9	11 101 001		FLDL2T	
D9	11 101 010		FLDL2E	
D9	11 101 011		FLDPI	
D9	11 101 100		FLDLG2	
D9	11 101 101		FLDLN2	
D9	11 101 110		FLDZ	
D9	11 110 000		F2XM1	
D9	11 110 001		FYL2X	
D9	11 110 010		FPTAN	
D9	11 110 011		FPATAN	
D9	11 110 100		FXTRACT	
D9	11 110 110		FDECSTP	
D9	11 110 111		FINCSTP	
D9	11 111 000		FPREM	
D9	11 111 001		FYL2XP1	
D9	11 111 010		FSQRT	
D9	11 111 100		FRNDINT	
D9	11 111 101		FSCALE	

Kodebytes 1	2	3 bis 6	Befehl	
DA	MD 000 R/M	**	FIADD	short integ
DA	MD 001 R/M	**	FIMUL	short integ
DA	MD 010 R/M	**	FICOM	short integ
DA	MD 011 R/M	**	FICOMP	short integ
DA	MD 100 R/M	**	FISUB	short integ
DA	MD 101 R/M	**	FISUBR	short integ
DA	MD 110 R/M	**	FIDIV	short integ
DA	MD 111 R/M	**	FIDIVR	short integ
DB	MD 000 R/M	**	FILD	short integ
DB	MD 010 R/M	**	FIST	short integ
DB	MD 011 R/M	**	FISTP	short integ
DB	MD 101 R/M	**	FLD	temp real
DB	MD 111 R/M	**	FSTP	temp real
DB	11 100 000		FENI	
DB	11 100 001		FDISI	
DB	11 100 010		FCLEX	
DB	11 100 011		FINIT	
DC	MD 000 R/M	**	FADD	long real
DC	MD 001 R/M	**	FMUL	long real
DC	MD 010 R/M	**	FCOM	long real
DC	MD 011 R/M	**	FCOMP	long real
DC	MD 100 R/M	**	FSUB	long real
DC	MD 101 R/M	**	FSUBR	long real
DC	MD 110 R/M	**	FDIV	long real
DC	MD 111 R/M	**	FDIVR	long real
DC	11 000 R/M		FADD	ST(i),ST
DC	11 001 R/M		FMUL	ST(i),ST
DC	11 010 R/M		FCOM	ST(i) +
DC	11 011 R/M		FCOMP	ST(i) +
DC	11 100 R/M		FSUB	ST(i),ST
DC	11 101 R/M		FSUBR	ST(i),ST
DC	11 110 R/M		FDIV	ST(i),ST
DC	11 111 R/M		FDIVR	ST(i),ST
DD	MD 000 R/M	**	FLD	long real
DD	MD 010 R/M	**	FST	long real
DD	MD 011 R/M	**	FSTP	long real
DD	MD 100 R/M	**	FRSTOR	mem
DD	MD 110 R/M	**	FSAVE	mem
DD	MD 111 R/M	**	FSTSW	mem
DD	11 000 R/M		FFREE	ST(i)
DD	11 001 R/M		FXCHG	ST(i) +
DD	11 010 R/M		FST	ST(i)
DD	11 011 R/M		FSTP	ST(i)
DE	MD 000 R/M	**	FIADD	word integ
DE	MD 001 R/M	**	FIMUL	word integ
DE	MD 010 R/M	**	FICOM	word integ
DE	MD 011 R/M	**	FICOMP	word integ
DE	MD 100 R/M	**	FISUB	word integ
DE	MD 101 R/M	**	FISUBR	word integ
DE	MD 110 R/M	**	FIDIV	word integ
DE	MD 111 R/M	**	FIDIVR	word integ

Kodebytes 1	2	3 bis 6	Befehl
DE	11 000 R/M		FADDP ST(i),ST
DE	11 001 R/M		FMULP ST(i),ST
DE	11 010 R/M		FCOMP ST(i) +
DE	11 011 001		FCOMPP
DE	11 100 R/M		FSUBP ST(i),ST
DE	11 101 R/M		FSUBRP ST(i),ST
DE	11 110 R/M		FDIVP ST(i),ST
DE	11 111 R/M		FDIVRP ST(i),ST
DF	MD 000 R/M	**	FILD word integ
DF	MD 010 R/M	**	FIST word integ
DF	MD 011 R/M	**	FISTP word integ
DF	MD 100 R/M	**	FBLD decimal
DF	MD 101 R/M	**	FILD long integ
DF	MD 110 R/M	**	FBSTP decimal
DF	MD 111 R/M	**	FISTP long integ
DF	11 000 R/M		FFREE ST(i) & Pop +
DF	11 001 R/M		FXCH ST(i) +
DF	11 010 R/M		FSTP ST(i) +
DF	11 011 R/M		FSTP ST(i) +
E0	IP-disp		LOOPNE/LOOPNZ shortlab
E1	IP-disp		LOOPE/LOOPZ shortlab
E2	IP-disp		LOOP shortlab
E3	IP-disp		JCXZ shortlab
E4	data		IN AL,port
E5	data		IN AX,port
E6	data		OUT port,AL
E7	data		OUT port,AX
E8	IP-disp-lo	IP-disp-hi	CALL nearproc
E9	IP-disp-lo	IP-disp-hi	JMP nearlab
EA	IP-lo	IP-hi,CS-lo,CS-hi	JMP farlab
EB	IP-disp		JMP shortlab
EC	data		IN AL,DX
ED	data		IN AX,DX
EE	data		OUT DX,AL
EF	data		OUT DX,AX
F0			LOCK
F1			–
F2			REPNE/REPNZ
F3			REP/REPE/REPNE
F4			HLT
F5			CMC
F6	MD 000 R/M	** , data	TEST reg8/mem8,data8
F6	MD 010 R/M	**	NOT reg8/mem8
F6	MD 011 11M	**	NEG reg8/mem8
F6	MD 100 R/M	**	MUL reg8/mem8
F6	MD 101 R/M	**	IMUL reg8/mem8
F6	MD 110 R/M	**	DIV reg8/mem8
F6	MD 111 R/M	**	IDIV reg8/mem8
F7	MD 000 R/M	** , data-lo,data-hi	TEST reg16/mem16,data16
F7	MD 010 R/M	**	NOT reg16/mem16
F7	MD 011 R/M	**	NEG reg16/mem16

Kodebytes 1	2	3 bis 6	Befehl
F7	MD 100 R/M	**	MUL reg16/mem16
F7	MD 101 R/M	**	IMUL reg16/mem16
F7	MD 110 R/M	**	DIV reg16/mem16
F7	MD 111 R/M	**	IDIV reg16/mem16
F8			CLC
F9			STC
FA			CLI
FB			STI
FC			CLD
FD			STD
FE	MD 000 R/M	**	INC reg8/mem8
FE	MD 001 R/M	**	DEC reg8/mem8
FF	MD 000 R/M	**	INC mem16
FF	MD 001 R/M	**	DEC mem16
FF	MD 010 R/M	**	CALL reg16/mem16
FF	MD 011 R/M	**	CALL mem32
FF	MD 100 R/M	**	JMP reg16/mem16
FF	MD 101 R/M	**	JMP mem32
FF	MD 110 R/M	**	PUSH mem16

Adreßmodus:

REG oder R/M	Registeroperand (REG oder R/M bei MD=11)	Speicheroperand ** (R/M bei MD≠11)
000	AX / AL / ST	[BX] + [SI] + **
001	CX / CL / ST(1)	[BX] + [DI] + **
010	DX / DL / ST(2)	[BP] + [SI] + **
011	BX / BL / ST(3)	[BP] + [DI] + **
100	SP / AH / ST(4)	[SI] + **
101	BP / CH / ST(5)	[DI] + **
110	SI / DH / ST(6)	[BP] + ** / DA (MD=00)
111	DI / BH / ST(7)	[BX] + **

Anmerkungen:

1. ** : bei MD=01 ein Displacementbyte
   bei MD=10 zwei Displacementbytes
   bei MD=00 / 11 kein Displacementbyte
   (außer MD=00,R/M=110: zwei Displacementbytes)
2. Die Kodes von D8 bis DF sind ESC-Kodes, die vom K1810WM87 dekodiert werden.
3. Alle fehlenden Kodes sind nicht definiert. Die mit + gekennzeichneten Kodes werden nicht vom Assembler generiert. Sie werden aber vom K1810WM87 interpretiert, falls er sie erkennt.

Literaturverzeichnis

/1/ Grafik, W.: Aufbau und Arbeitsweise von 16-Bit-Mikroprozessoren. Reihe Automatisierungstechnik 226.
Berlin: VEB Verlag Technik 1987

/2/ iAPX 86,88 User's Manual. Santa Clara: Intel Corp. 1981

/3/ Technische Bedingungen: TU II-85 KM1810WM86: 0.348.800-07TU

/4/ A7100, Arbeitsplatzcomputer Bd.1: Rechner und Geräte; Betriebsdokumentation.
Dresden: VEB Robotron Elektronik Dresden 1986

/5/ Anleitung für den Assemblerprogrammierer; Anwenderdokumentation System DCP3.2.
Karl-Marx-Stadt: VEB Robotron Buchungsmaschinenwerk 1987

/6/ Scanlon, L.J.: IBM-PC Assembly Language; A Guide For Programmers.
Bowie: Prentice-Hall 1986

/7/ Rector, R.; Alexy, G.: The 8086 Book.
Berkeley: Osborn/McGraw-Hill 1980

Sachwörterverzeichnis

Addition 63ff., 172f.
Adresse, effektive 13, 47ff., 59f., 105, 162f.
-, logische 29, 31
-, physische 22, 31f., 44, 48
Adressierung 13, 26, 28, 33f., 47ff., 59, 80, 99, 101, 163
Adreßmodusbyte 54
Akkumulator 18, 53, 62, 70ff.,
Arithmetikoperation 56, 63ff., 146
Array 50ff.
Assembler 14, 48ff., 81, 87f., 102, 143, 147ff., 166
-anweisung 57, 146
-direktive 89, 147
-sprache 14, 45, 143ff.
ASSUME 150
Ausführungseinheit 20ff., 47
Ausführungszeit 18, 39f., 46, 102, 105f., 153

Basisadresse 13, 18, 31ff., 47
BCD 20, 64ff., 146, 157f., 171
Befehlswarteschlange 22, 29, 41, 87
Befehlszeiger 13, 19, 22, 89f., 161f.
Bias 158
Binärzahl 56, 63, 66ff., 79, 153, 156
Bitverschiebebefehl 18, 77
BIU 21ff., 42f., 106
Bus 15, 25, 29, 41f., 100
-anforderung 41ff., 101
-blockierung 42, 101
-interfaceeinheit 20ff.
-zyklus 16, 22ff., 39, 46, 106, 162
Byte Integer 63
Byteübertragung 29, 34, 106

Compiler 14, 33, 143, 155
Coprozessor 14, 42f., 100f., 153ff.

Datenregister 13, 18, 22
Datensegment 13, 19, 31, 34, 48ff.
Datentransfer 14, 24, 42, 56f.
Datenvereinbarung 148
Dezimalzahl 56, 63f., 66ff., 72f., 153
Direktive 14, 146ff.
Displacement 13, 44, 47ff., 55, 92
Division 36, 63ff., 72f., 79, 166, 172ff.
-sfehler 35ff., 72f.
Doppelwort 46, 65f., 72ff., 87
-variable 60, 87f.
-zeiger 35, 85, 102, 161
DUP 149

Ein-/Ausgabe, speicheradressierte 34
-befehl 56, 62
-operation 18, 23, 34
-port 34, 44, 53, 62
-prozessor 14, 42
Einzelschrittbetrieb 36, 83
Emulator 156, 166, 181
EQU 148
ESC-Befehl 14, 43, 155f., 162, 165, 168f.
ESC-Kode 100, 155, 162, 181
EU 21ff., 43, 106
even bank 28
Exception 159ff., 166ff., 179
-Handler 167ff.
-Pointer 161, 179f.
Exklusiv-ODER 77
Exponent 156ff., 164f., 175
Extradatensegment 13, 18f., 81
EXTRN 152

Flag 19f., 38ff., 41, 61f., 65f., 74f., 91

Genauigkeit 156, 159ff., 167
Gleitkommaregister 156, 158
Gleitkommazahl 156f.

Halt-Zustand 42f., 100
high bank 28

Indefinit-Kode 165
Indexregister 13, 18, 44, 47, 51,
Initialisierung 41, 169f.
Integer 63, 156ff., 171f.
Intersegment 87

Interrupt 37ff., 83, 97ff., 168
-anerkennung 23, 38ff.
-anforderung 20, 35ff., 42f., 154, 167f.
-befehl 38, 97f.
-prozedur 35ff., 61, 83, 97f., 161, 167ff., 179
-vektortabelle 35ff., 97
Intrasegment 88

Koderegister 22
Kodesegment 13, 19, 22, 31, 35, 87f.
Konstante 46, 67ff., 147f., 178

Leseoperation 21, 23f.
Lock-Präfix 42f., 58, 98
Logikoperation 75
Long Integer 157, 172
Long Real 157f., 171f.

Mantisse 156
Marke 87ff., 146f., 151
Maschinenbefehl 46, 53ff., 89, 91f., 102, 143, 146f.
Maschinenkode 49, 102, 143
Maschinensprache 143
Matrix 52
Maximum-Mode 15, 23, 42
Mehrrechnersystem 25, 42
Minimum-Mode 15, 23, 42
Mnemonik 77, 81, 102, 146f.
Multiplikation 18, 63ff., 70f., 77, 172ff.
Multiprozessorsystem 14

NAN-Kode 165
NICHT 75f.
NMI 37f., 169
Numerikcoprozessor 14, 42, 146, 149, 153ff.

Objektdatei 144f.
Objektmodul 87, 144f.
ODER 75
odd bank 28
Offset 31ff., 47, 80
-adresse 18f., 31f.
Operand, denormaler 159f., 167
Operation, arithmetische 14, 56, 153
-, logische 56, 74f.
-, ungültige 159f., 167

Parameter 33, 39, 59, 90, 163
Portadresse 18, 34, 53, 62
Programmiersprache, höhere 14, 143
Programmlaufzeit 22, 102, 106
Programmstatuswort 13, 19, 61
Programmtransferbefehl 56, 87
Programmverzweigung 22, 33, 35, 56
Prozedur 35ff., 88f., 97, 151
PSW 19, 36, 39
PTR 150
PUBLIC 152

Quelldatei 144
Quellmodul 144
Quellstring 80, 84ff.

Real 146, 157
Repeat-Präfix 38, 80ff.
Reset 41
Rotationsbefehl 77

Schleifenbefehl 18, 88, 95f.
Schreiboperation 21, 23f.
Segment 13, 19, 31, 48, 88, 150
-basisadresse 13, 48
-override-Präfix 48ff., 80
-register 18, 22, 31f., 38, 44, 47f., 54, 57f., 60, 101
-überschreibung 48, 53
-zuweisung 49, 150
Semaphor 42, 58
Short Integer 157, 172
Short Real 157f., 171
Signifikand 156, 164f., 175
Sprungbefehl 23, 66, 75, 91
Stapel 19, 33, 39, 59, 61, 88, 97f.
-operation 33, 47
-register 158ff., 171ff.
-segment 13, 18f., 31, 33, 49ff.
-spitze 158, 163, 171ff.
-zeiger 33, 59, 61, 158, 160, 163, 171ff.
Statussignale 15
Statuswort 160, 163, 179f.
Steuerwort 159, 179f.

Stringbefehl 56, 80ff.
Stringoperation 18f., 53, 60, 81ff., 101
Struktur 13, 50ff.
Subtraktion 63ff., 68ff., 172f.
Synchronisation 56, 100, 154
Systembus 25

TAG-Wort 160, 164, 179f.
Temporary Real 157f., 163, 171ff.

Überlauf 159f., 167
UND 75
Unterschreitung 159f., 167
Unterprogramm 33, 36f., 59ff., 87ff., 156, 163, 170

Variable 48ff., 60, 148f., 162
Verbindeprogramm 144

Wert, spezieller 163
-, unnormaler 164
Wiederholungspräfix 18
Word Integer 63, 157, 172
Wortübertragung 29, 33, 106

Zahl, denormierte 164
-, nichtnormierte 164
-, unnormale 164
Zeichenkette 56, 80, 146
Zielstring 81, 84ff.
Zugriffszeit 24
Zweierkomplement 63, 156

Das Herz des ESER-PC ist der Schaltkreis K1810WM86, ein Mikroprozessor der 16-Bit-Generation. Wie ist er aufgebaut, wie arbeitet er mit Speicher- und Ein-/Ausgabebaugruppen zusammen, und wie wird er programmiert? Antwort auf diese Fragen gibt der vorliegende Band, wobei die Assemblerprogrammierung im Mittelpunkt steht. Ergänzend wird der Numerikcoprozessor des K 1810-Systems vorgestellt.

ISBN 3-341-00704-0